digital

processor handbook

pdp11/45

digital pdp11/45 processor handbook 1972

digital

pdp11/45

processor handbook

digital equipment corporation

Copyright 1971 by
Digital Equipment Corporation
PDP, UNIBUS, DEC are registered trademarks
of Digital Equipment Corporation

DEC Typesetting
This Handbook was "typed" and edited with the aid of a DECsystem 10 time-sharing system. Type was set via a DEC typesetting system.

digital

The PDP-11 is a family of upward-compatible computer systems. We believe that these systems represent a significant departure from traditional methods of computer design.

The initial design step was the development of a totally new language, notation, and theory of computers called the Instruction Set Processor (ISP). This language provides a concise and powerful generalized method for defining an arbitrary computer system and its operation. Along with the development of ISP, a PDP-10 program was written for simulating the operation of any computer system on the basis of its ISP description. With the aid of ISP and the machine simulation program, benchmark comparison tests were run on a large number of potential computer designs. In this manner it was possible to evaluate a variety of design choices and compare their features and advantages, without the time and expense of actually constructing physical prototypes.

Since the main design objective of the PDP-11 was to optimize total system performance, the interaction of software and hardware was carefully considered at every step in the design process. System programmers continually evaluated the efficiency of the code which would be produced by the system software, the ease of coding a program, the speed of real-time response, the power and speed that could be built into a system executive, the ease of system resource management, and numerous other potential software considerations.

The current PDP-11 Family is the result of this design effort. We believe that its general purpose register and UNIBUS organization provides unparalleled power and flexibility. This design is the basis for our continuing commitment to further PDP-11 product development.

Thus the PDP-11 Family is at once a new concept in computer systems, and a tested and tried system. The ultimate proof of this new design approach has come from the large and rapidly increasing number of PDP-11 users all around the world.

Kenneth H. Olsen
President,
Digital Equipment Corporation

CONTENTS

CHAPTER 1 - INTRODUCTION 1

1.1 PDP-11 FAMILY 1
1.2 GENERAL CHARACTERISTICS 1
1.3 PERIPHERALS AND OPTIONS 5
1.4 SOFTWARE 6

CHAPTER 2 - SYSTEM ARCHITECTURE 9

2.1 INTRODUCTION 9
2.2 THE UNIBUS 10
2.3 CENTRAL PROCESSOR 11
2.4 FLOATING POINT PROCESSOR 14
2.5 MEMORY 14
2.6 SYSTEM INTERACTION 18
2.7 PROCESSOR TRAPS 22
2.8 MULTI-PROGRAMMING 24

CHAPTER 3 - ADDRESSING MODES 25

3.1 SINGLE OPERAND ADDRESSING 26
3.2 DOUBLE OPERAND ADDRESSING 26
3.3 DIRECT ADDRESSING 28
3.4 DEFERRED ADDRESSING 34
3.5 USE OF PC 36
3.6 USE OF STACK POINTER 40

CHAPTER 4 - INSTRUCTION SET 43

4.1 INTRODUCTION 43
4.2 INSTRUCTION FORMATS 44
4.3 BYTE INSTRUCTIONS 46
4.4 SINGLE OPERAND INSTRUCTIONS 47
4.5 DOUBLE OPERAND INSTRUCTIONS 68
4.6 PROGRAM CONTROL INSTRUCTIONS 81
4.7 MISCELLANEOUS INSTRUCTIONS 122
4.8 CONDITION CODE OPERATORS 130

CHAPTER 5 - ADVANCED PROGRAMMING TECHNIQUES 131

5.1 THE STACK 131
5.2 SUBROUTINE LINKAGE 135
5.3 INTERRUPTS 139
5.4 REENTRANCY 143
5.5 POSITION INDEPENDENT CODE 145
5.6 RECURSION 146
5.7 CO-ROUTINES 146

CHAPTER 6 - MEMORY MANAGEMENT 147

6.1 BASIC ADDRESSING LOGIC 147
6.2 VIRTUAL ADDRESSING 148
6.3 INTERRUPT CONDITIONS UNDER SEGMENTATION CONTROL 149
6.4 CONSTRUCTION OF A PHYSICAL ADDRESS 149
6.5 SEGMENTATION REGISTERS 151
6.6 SEGMENTATION FAULT RECOVERY REGISTERS 155
6.7 SEGMENTATION EXAMPLES 159
6.8 TRANSPARENCY 163
6.9 SEGMENTATION REGISTER MAP 164

CHAPTER 7 - FLOATING POINT PROCESSOR 167

7.1 INTRODUCTION 167
7.2 OPERATION 167
7.3 ARCHITECTURE 168
7.4 FLOATING POINT DATA FORMATS 169
7.5 FPP STATUS REGISTER 170
7.6 FEC REGISTER 172
7.7 FPP INSTRUCTION ADDRESSING 172
7.8 INSTRUCTION TIMING 172
7.9 FPP INSTRUCTIONS 173

CHAPTER 8 - SYSTEM OPERATOR'S CONSOLE 197

8.1 CONSOLE ELEMENTS 197
8.2 SYSTEM POWER SWITCH 198
8.3 CPU STATE INDICATORS 199
8.4 ADDRESS DISPLAY REGISTER 200
8.5 ADDRESSING ERROR DISPLAY 200
8.6 DATA DISPLAY REGISTER 201
8.7 SWITCH REGISTER 201
8.8 CONTROL SWITCHES 201

APPENDIXES

APPENDIX A - INSTRUCTION SET PROCESSOR (ISP) 205

APPENDIX B - INSTRUCTION TIMING 219

APPENDIX C - MEMORY MAP 233

APPENDIX D - PROGRAM INTERRUPT REQUESTS 239

APPENDIX E - MEMORY PARITY 241

INDEXES

GENERAL INDEX 243

INSTRUCTION INDEX 244

CHAPTER 1

INTRODUCTION

The PDP-11/45 is a powerful 16-bit computer representing the large computer end of the PDP-11 family of computers. It is designed as a powerful computational tool for high-speed real-time applications and for large multi-user, multi-task applications requiring up to 124K words of addressable memory space. It will operate with solid state and core memories, and includes many features not normally associated with 16-bit computers. Among its major features are a fast central processor with choices of 300 or 450 nanosecond memory, an advanced Floating Point Processor, and a sophisticated memory management scheme.

1.1 THE PDP-11 FAMILY

The PDP-11 family includes several processors, a large number of peripheral devices and options, and extensive software. PDP-11 machines are architecturally similar and hardware and software upwards compatible, although each machine has some of its own characteristics. New PDP-11 systems will be compatible with existing family members. The user can choose the system which is most suitable to his application, but as needs change or grow he can easily add or change hardware. The major characteristics of PDP-11 family computers are summarized in Table 1-1 at the end of this chapter.

1.2 GENERAL CHARACTERISTICS

1.2.1 The UNIBUS

All computer system components and peripherals connect to and communicate with each other on a single high-speed bus known as the UNIBUS -- the key to the PDP-11's many strengths. Since all system elements, including the central processor, communicate with each other in identical fashion via the UNIBUS, the processor has the same easy access to peripherals as it has to memory.

With bidirectional and asynchronous communications on the UNIBUS, devices can send, receive and exchange data independently without processor intervention. For example, a CRT display can refresh itself from a disk file while the central processor unit (CPU) attends to other tasks. Because it is asynchronous the UNIBUS is compatible with devices operating over a wide range of speeds.

Device communications on the UNIBUS are interlocked. For each command issued by a "master" device, a response signal is received from a "slave" completing the data transfer. Device-to-device communication is completely independent of physical bus length and the response times of master and slave devices. Interfaces to the UNIBUS are not time-dependent; there are no pulse-width or rise-time restrictions to worry about. The maximum transfer rate on the UNIBUS is one 16-bit word every 400 nanoseconds, or 2,500,000 words per second.

Input/output (I/O) devices transferring directly to or from memory are given highest priority, and may request bus mastership and "steal" bus cycles during instruction operations. The processor resumes operation immediately after the memory transfer. Multiple devices can operate simultaneously at maximum direct

memory access (DMA) rates by stealing bus cycles. The UNIBUS is further explained in Paragraph 2.2, Chapter 2, and is covered in considerable detail in the PDP-11 Peripherals and Interfacing Handbook.

1.2.2 Central Processor

The central processor, connected to the UNIBUS as a subsystem, controls the allocation of the UNIBUS for peripherals and performs arithmetic and logic operations and instruction decoding. It contains multiple high-speed general-purpose registers which can be used as accumulators, pointers, index registers, or as autoindexing pointers in autoincrement or autodecrement modes. The processor can perform data transfers directly between I/O devices and memory without disturbing the registers; does both single-and double-operand addressing; handles both 16-bit word and 8-bit byte data; and, by using its dynamic stacking technique, allows nested interrupts and automatic reentrant subroutine calling.

Instruction Set

The instruction complement uses the flexibility of the general-purpose registers to provide over 400 powerful hard-wired instructions -- the most comprehensive and powerful instruction repertoire of any computer in the 16-bit class. Unlike conventional 16-bit computers, which usually have three classes of instructions - memory reference instructions, operate or AC control instructions, and I/O instructions - all operations in the PDP-11 are accomplished with one set of instructions. Since peripheral device registers can be manipulated as flexibly as core memory by the central processor , instructions that are used to manipulate data in core memory may be used equally well for data in peripheral device registers. For example, data in an external device register can be tested or modified directly by the GPU without bringing it into memory or disturbing the general registers. One can add data directly to a peripheral device register, or compare contents with a mask and branch. Thus all PDP-11 instructions can be used to create a new dimension in the treatment of computer I/O and the need for a special class of I/O instructions is eliminated. PDP-11/45 instructions are described in Chapter 4.

The following example contrasts the rotate operation in the PDP-11 with a similar operation in a conventional computer:

	PDP-11 Approach		**Conventional Approach**
ROR A	; rotate contents of memory location A right one place	LDA A	;load contents of memory location A into AC
		ROT	; rotate contents of AC right one place
		STA A	; store contentents of AC in location A

The basic order code of the PDP-11 uses both single and double operand address instructions for words or bytes. The PDP-11 therefore performs very efficiently in

Stacks
In a PDP-11, a stack is a temporary data storage area which allows a program to make efficient use of frequently accessed data. The stack is used automatically by program interrupts, subroutine calls, and trap instructions. When the processor is interrupted, the central processor status word and the program counter are saved (pushed) onto the stack area, while the processor services the interrupting device. A new status word is then automatically acquired from an area in core memory which is reserved for interrupt instructions (vector area). A return from the interrupt instruction restores the original processor status and returns control to the interrupted program without software intervention. Stacks are explained in Chapter 5.

Direct Memory Access
All PDP-11s provide for direct access to memory. Any number of DMA devices may be attached to the UNIBUS. Maximum priority is given to DMA devices thus allowing memory data storage or retrieval at memory cycle speeds. Latency is minimized by the organization and logic of the UNIBUS which samples requests and priorities in parallel with data transfers.

Power Fail and Restart
Power fail and restart, not only protects memory when power fails, but also allows the user to save the existing program location and status (including all dynamic registers), thus preventing harm to devices, and eliminating the need for reloading programs. Automatic restart is accomplished when power returns to safe operating levels, enabling remote or unattended operations of PDP-11 systems.

1.2.3 Memories
Memories with different ranges of speeds and various characteristics can be freely mixed and interchanged in a single PDP-11 system. Thus as memory needs expand and as memory technology grows, a PDP-11 can evolve, with none of the growing pains and obsolescence associated with conventional computers. See Paragraph 2.6, Chapter 2.

1.2.4 Floating Point Processor
An advanced-design Floating Point Processor functions as an integral part of the PDP-11/45 central processor. Floating point instructions overlap CPU instructions and can continue without CPU intervention, leaving the CPU free to execute other instructions. Floating Point Processor instructions are described in Chapter 7.

1.2.5 Memory Segmentation
PDP-11/45 segmentation is an advanced memory extension, relocation and protection feature which will:

extend memory space from 28K to 124K words

allow efficient segmentation of core for multi-user environments

provide effective protection of memory segments in multi-user environments.

Segmentation is explained in Chapter 6.

1.3 PERIPHERALS OPTIONS

Digital Equipment Corporation designs and manufactures many of the peripheral devices offered with PDP-11s. As a designer and manufacturer of peripherals, DEC can offer extremely reliable equipment, lower prices, more choices, and quantity discounts.

Many processor, input/output, memory, bus, and storage options are available. These devices are explained in detail in the Peripherals and Interfacing Handbook.

1.3.1 I/O Devices

All PDP-11 systems are available with Teletypes as standard equipment. However, their I/O capabilities can be increased with high speed paper tape readers-punches, line printers, card readers or alphanumeric display terminals. The LA30 DECwriter, a totally DEC designed and built teleprinter, can serve as an alternative to the Teletype. It has several advantages over standard electromechanical typewriter terminals, including higher speed, fewer mechanical parts and very quiet operation.

PDP-11 terminals include:

DECterminal alphanumeric display

DECwriter teleprinter

High-speed line printers

High-speed paper tape reader punch

Teletypes

Card readers

1.3.2 Storage Devices

Storage devices range from convenient, small-reel magnetic tape units to mass storage magnetic tapes and disk memories. With the UNIBUS, a large number of storage devices, in any combination, may be connected to a PDP-11 system. TU56 DECtapes, highly reliable tape units with small tape reels, designed and built by DEC, are ideal for applications with modest storage requirements. Each DECtape provides storage for 147K 16 bit words. For applications which require handling of large volumes of data, DEC offers the industry compatible TU10 Magtape.

Disk storage devices include fixed head disk units and moving-head removable cartridge and disk pack units. These devices range from the 65K RS64 DECdisk memory, to the RP02 Disk Pack system which can store up to 93.6 million words. PDP-11 storage devices include:

DECtape

Magtape

RS64 65K-256K word fixed head disk

RF11 256K-2M word fixed head disk

RK03 1-2M word moving-head disk

RP02 10M word moving head disk

1.3.3 Bus Options

Several options (bus switches, bus extenders) are available for extending the UNIBUS or for configuring multi-processor or shared-peripheral systems.

1.4 SOFTWARE

Extensive software, consisting of disk and paper tape systems, is available for PDP-11 Family systems. The larger the PDP-11 configuration, the larger and more comprehensive the software package that comes with it.

1.4.1 Disk Operating System Software

The Disk Operating System software includes:

Text Editor (ED-11)

Relocatable Assembler (PAL-11R)

Linker (LINK-11)

File Utilities Packages (PIP)

On Line Debugging Technique (ODT-11R)

Librarian (LIBR-11)

1.4.2 Higher Level Languages

PDP-11 users needing an interactive conversational language can use BASIC which can be run on the paper tape software system with only 4.096 words of core memory. A multi-user extension of BASIC is available so up to eight terminal users can access a PDP-11 with only 8K of core.

RSTS-11

The PDP-11 Resource Timesharing System (RSTS-11) with BASIC-PLUS, an enriched version of BASIC, is available for up to 16 terminal users.

FORTRAN

PDP-11 FORTRAN is an ANSI-standard Fortran IV compiler with elements that provide easy compatability with IBM 1130 FORTRAN.

Paper tape software is available on systems without disks.

1.5 DATA COMMUNICATIONS

The advanced architecture of PDP-11 family machines makes them ideal for use in data communications applications. For example the UNIBUS performs like a multiplexer and multiple single-line interfaces can be added without special multiplexing hardware: byte handling, the key to communications applications, is accomplished easily and efficiently by the PDP-11. To provide total systems capability in the communications area DEC has developed a full line of communications hardware and communications-oriented software.

1.6 DATA ACQUISITION AND CONTROL

The PDP-11, modular process interfaces, and special state-of-the art software (RSX-11C real-time executive) combine to provide efficient, low-cost and reliable systems for industrial data acquisition and control (IDACS) applications. IDACS-11 hardware is described in the Peripherals and Interfacing Handbook.

TABLE 1-1
PDP-11 Family Computers

	PDP-11/05	PDP-11/15	PDP-11/20	PDP-11/45
Central Processor	KD11	KC11	KA11	KB11
General Purpose Registers	8	8	8	16
Instructions	Basic Set	Basic Set	Basic Set	Basic Set and: MUL, DIV, XOR. ASH, ASHC, MARK, SXT, SOB, SPL, RTT MTPX, MFPX
Segmentation Hardware	No	No	No.	Optional
Stacks	Yes	Yes	Yes	Yes
Stack overflow Detection	Fixed	Fixed	Fixed	Variable
Automatic Priority Interrupts	Single-Line, Multi-Level	Single-Line Multi-Level (four line optional)	Multi-Line Multi-Level	Multi-Line Multi-Level Plus 7 Software Levels
Overlapped Instructions	No	No	No	Yes
Extended Arithmetic	Option	Option	Option	Standard
Floating Point	Software	Software	Software	Internal to CPU
Basic Memory	Core	Core	Core	Core, MOS or Bipolar

CHAPTER 2

SYSTEM ARCHITECTURE

2.1 INTRODUCTION

The PDP-11/45 is a medium scale general purpose computer designed around the basic architecture of all PDP-11 family machines.

The Central Processing Unit has a cycle time of 300 nsec and performs all arithmetic and logical operations required in the system. A Floating Point Processor (described in Chapter 7) mounts integrally into the Central Processor as does a Memory Segmentation Unit which provides a full memory management facility through relocation and protection (described in Chapter 6).

The PDP-11/45 hardware has been optimized towards a multi-programming environment and the processor therefore operates in three modes (Kernel, Supervisor, and User) and has two sets of General Registers.

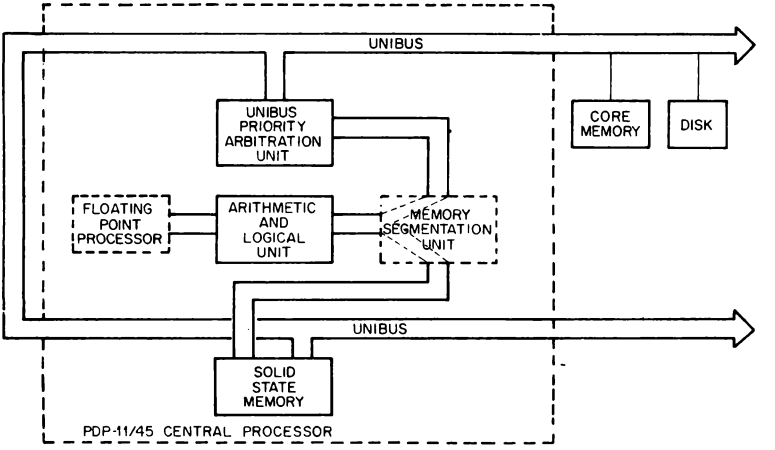


Figure 2-1 PDP-11/45 System Block Diagram

The PDP-11/45 communicates with its options through a bidirectional, asynchronous bus, the UNIBUS.

2.2 THE UNIBUS

All devices are connected through hardware registers to the UNIBUS.

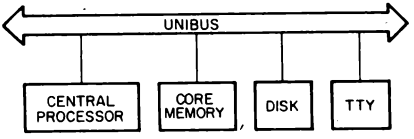


Figure 2-2 The UNIBUS

Any device (except memory) can dynamically request the UNIBUS to transfer information to another using a scheme based on real and simulated core locations. All device registers are located in the uppermost 4K words of address space (124K-128K). Thus, the Central Processor can look on its peripherals as if they were locations in memory with special properties, and operate on them using the same set of instructions it uses to operate on memory.

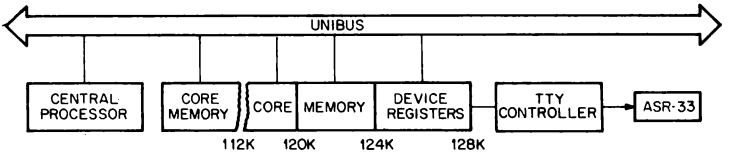


Figure 2-3 Location of Device Registers

The UNIBUS provides the communications path for address, data, and control information for all devices on the bus through its bidirectional lines. Therefore the same device registers can be used for both input and output functions.

Devices communicate on the UNIBUS in a master-slave relationship. During any bus operation, one device has control of the bus. This device, called the master, controls the bus when communicating with another, called the slave.

The relationship is dynamic, thus the Central Processor as master could send control information to a disk (slave) which then could obtain the bus as a master to communicate with memory, the slave.

The UNIBUS is used by the processor and all I/O devices. A priority structure determines which device has control of the bus at any given instant of time. Therefore, every device capable of becoming bus master has an assigned priority; and when two devices request the bus at the same time, the device with the higher priority will receive control first.

Communication on the UNIBUS is interlocked between devices. For each control signal issued by the master, there is a response from the slave; thus, communication is independent of physical bus length and the response time of the master and slave devices. The maximum transfer rate on the UNIBUS is one 16-bit word every 400 nsec or 2.5 million 16-bit words per second. The UNIBUS is fully described in the PDP-11 Peripherals and Interfacing Handbook.

2.3 CENTRAL PROCESSOR

The PDP-11/45 performs all arithmetic and logical operations required in the system. It also acts as the arbitration unit for UNIBUS control by regulating bus requests and transferring control of the bus to the requesting device with the highest priority.

The central processor contains arithmetic and control logic for a wide range of operations. These include high-speed fixed point arithmetic with hardware multiply and divide, extensive test and branch operations, and other control operations. It also provides room for the addition of the high-speed Floating Point Processor, and Memory Segmentation Unit.

The machine operates in three modes: Kernel, Supervisor, and User. When the machine is in Kernel mode a program has complete control of the machine; when the machine is in any other mode the processor is inhibited from executing certain instructions and can be denied direct access to the peripherals on the system. This hardware feature can be used to provide complete executive protection in a multi-programming environment.

The central processor contains 16 general registers which can be used as accumulators, index registers, or as stack pointers. Stacks are extremely useful for nesting programs, creating re-entrant coding, and as temporary storage where a Last-In First-Out structure is desirable. A special instruction "MARK"is provided to further facilitate re-entrant programming and is described in Chapter 5. One of the general registers is used as the PDP-11/45's program counter. Three others are used as Processor Stack Pointers, one for each operational mode.

The CPU is directly connected to the high-speed memories as well as to the general purpose registers and the UNIBUS and UNIBUS Priority Arbitration Unit.

Figure 2-4 illustrates the data paths in the CPU.

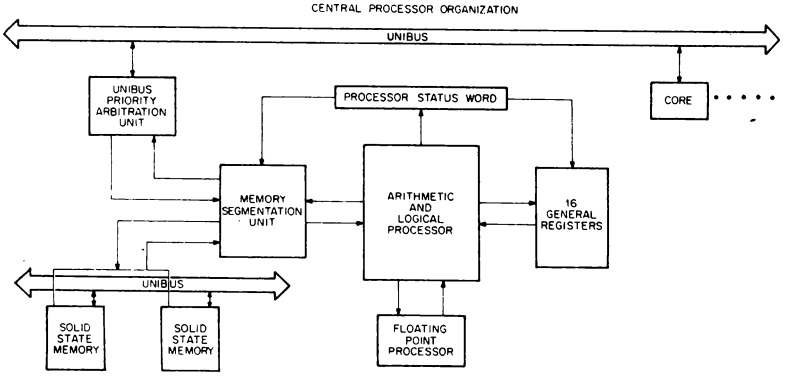


Figure 2-4 Central Processor Data Paths

The CPU performs all of the computer's computation and logic operations in a parallel binary mode through step by step execution of individual instructions. The instructions are stored in either core or solid state memory.

2.3.1 General Registers

The general registers can be used for a variety of purposes; the uses varying with requirements. The general registers can be used as accumulators, index registers, autoincrement registers, autodecrement registers, or as stack pointers for temporary storage of data. Chapter 3 on Addressing describes these uses of the general registers in more detail. Arithmetic operations can be from one general register to another, from one memory or device register to another, or between memory or a device register and a general register.

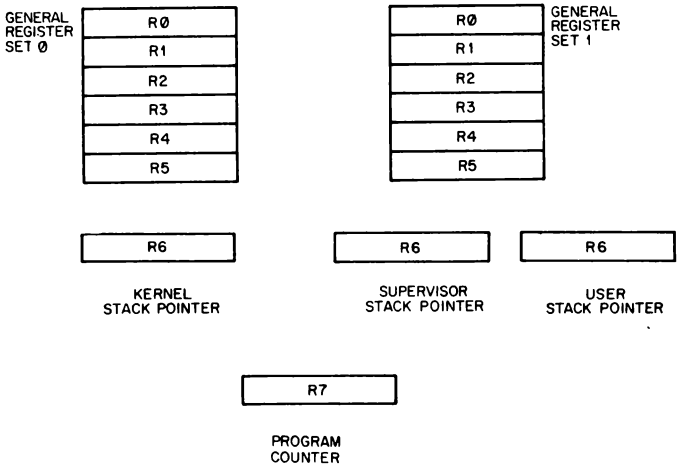


Figure 2-5 The General Registers

R7 is used as the machine's program counter (PC) and contains the address of the next instruction to be executed. It is a general register normally used only for addressing purposes and not as an accumulator for arithmetic operations.

The R6 register is normally used as the Processor Stack Pointer indicating the last entry in the appropriate stack (a common temporary storage area with "Last-in First-Out"characteristics). (For information on the programming uses of stacks, please refer to Chapter 5). The three stacks are called the Kernel Stack, the Supervisor Stack, and the User Stack. When the Central Processor is operating in Kernel mode it uses the Kernel Stack, in Supervisor mode, the Supervisor Stack, and in User mode, the User Stack. When an interrupt or trap occurs, the PDP-11/45 automatically saves its current status on the Processor Stack selected by the service routine. This stack-based architecture facilitates reentrant programming.

The remaining 12 registers are divided into two sets of unrestricted registers, R0-R5. The current register set in operation is determined by the Processor Status Word.

The two sets of registers can be used to increase the speed of real-time data handling or facilitate multi-programming. The six registers in General Register Set 0 could each be used as an accumulator and/or index register for a real-time task or device, or as general registers for a Kernel or Supervisor mode program. General Register Set 1 could be used by the remaining programs or User mode programs. TheSupervisorcan therefore protect its general registers and stack from User programs, or other parts of the Supervisor.

2.3.2 Processor Status Word

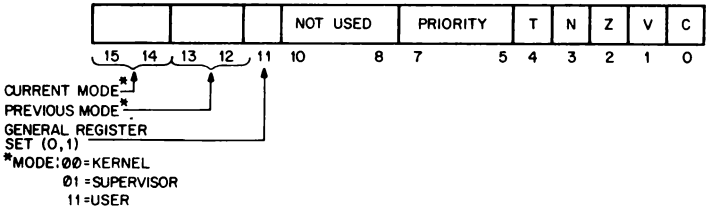


Figure 2-6 Processor Status Word

The Processor Status Word, located at location 777776, contains information on the current status of the PDP-11/45. This information includes the register set currently in use; current processor priority; current and previous operational modes; the condition codes describing the results of the last instruction; and an indicator for detecting the execution of an instruction to be trapped during program debugging.

Modes

Mode information includes the present mode, either User, Supervisor, or Kernel (bits 15, 14); the mode the machine was in prior to the last interrupt or trap (bits 13, 12); and which register set (General Register Set 0 or 1) is currently being used (bit 11).

The three modes permit a fully protected environment for a multi-programming system by providing the user with three distinct sets of Processor Stacks and Memory Segmentation Registers for memory mapping. In all modes except Kernel a program is inhibited from executing a "HALT" instruction and the processor will trap through location 4 if an attempt is made to execute this instruction. Furthermore, the processor will ignore the "RESET" and "SPL" instructions. In Kernel mode, the processor will execute all instructions.

A program operating in Kernel mode can map users' programs anywhere in core and thus explicitly protect key areas (including the device registers and the Processor Status Word) from the User operating environment.

Processor Priority

The Central Processor operates at any one of eight levels of priority, 0-7. When the CPU is operating at level 7 an external device cannot interrupt it with a request for service. The Central Processor must be operating at a lower priority than the priority of the external device's request in order for the interruption to take effect. The current priority is maintained in the processor status word (bits

5-7). The 8 processor levels provide an effective interrupt mask, which can be dynamically altered through use of the Set Priority Level (SPL) instruction which is described in Chapter 4 and which can only be used by the Kernel. This instruction allows a Kernel mode program to alter the Central Processor's priority without affecting the rest of the Processor Status Word.

Condition Codes
The condition codes contain information on the result of the last CPU operation. They include; a carry bit (C), which is set by the previous operation if the operation caused a carry out of its most significant bit; a negative bit (N) set if the result of the previous operation was negative; a zero bit (Z), set if the result of the previous operation was zero; and an overflow bit (V), set if the result of the previous operation resulted in an arithmetic overflow.

Trap
The trap bit (T) can be set or cleared under program control. When set, a processor trap will occur through location 14 on completion of instruction execution and a new Processor Status Word will be loaded. This bit is especially useful for debugging programs as it provides an efficient method of installing breakpoints.

Interrupts and trap instructions both automatically cause the previous Processor Status Word and Program Counter to be saved and replaced by the new values corresponding to those required by the routine servicing the interrupt or trap. The user can, thus, cause the central processor to automatically switch modes (context switching), register sets, alter the CPU's priority, or disable the Trap Bit whenever a trap or interrupt occurs.

2.3.3 Stack Limit Register
All PDP-11's have a Stack Overflow Boundary at location 400$_8$. The Kernel Stack Boundary, in the PDP-11/45 is a variable boundary set through the Stack Limit Register found at location 777774.

Once the Kernel stack exceeds its boundary, the Processor will complete the current instruction and then trap to location 4 (Yellow or Warning Stack Violation). If, for some reason, the program persists beyond the 16-word limit, the processor will abort the offending instruction, set the stack pointer (R6) to 4 and trap to location 4 (Red or Fatal Stack Violation). A description of these traps is contained in Appendix C.

2.4 FLOATING POINT PROCESSOR
The PDP-11/45 Floating Point Processor fits integrally into the Central Processor. It provides a supplemental instruction set for performing single and double precision floating point arithmetic operations and floating-integer conversions in parallel with the CPU. It is fully described in Chapter 7.

2.5 MEMORY
Memory is the primary storage medium for instructions and data. Three types are available for the PDP-11/45:

SOLID STATE:

Bipolar Memory with a cycle time of 300nsec.

MOS Memory with a cycle time of 450 nsec.

CORE:

Magnetic Core Memory with a cycle time of 850 ns, access at 350 ns (450 ns at the UNIBUS).

Any system can be expanded from the basic 4,096 words to 126,976 words in increments of 4,096 words. The system can be configured with various mixtures of the three types of memory, with a maximum limit of 32,768 words of Solid State Memory.

2.5.1 Solid State Memory

The Central Processor communicates directly with the MOS and Bipolar memories through a very high speed data path which is internal to the PDP-11/45 processor system. The CPU can control up to two independent Solid State Memory controllers, each of which can have from one to four 4,096 word increments of MOS memory (16,384 words) per controller, or one 4,096 word increment of Bipolar memory per controller. Each controller can handle MOS or Bipolar memory but not a mixture of the two. The user can therefore have a total of 32K of MOS, or 8K of Bipolar, or 16K of MOS and 4K of Bipolar.

Each controller has dual ports and provides one interface to the CPU and another to a second UNIBUS.

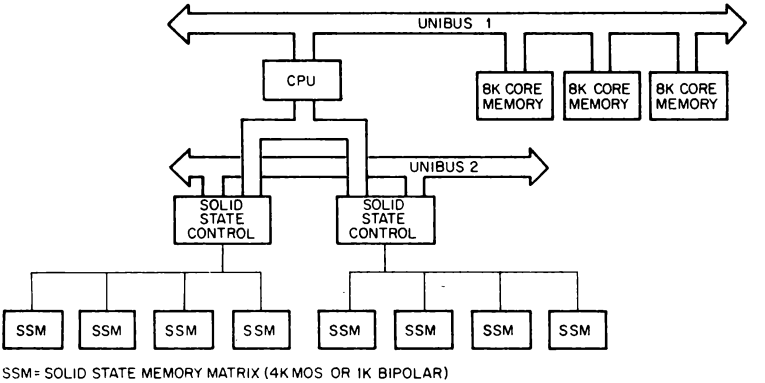


Figure 2-7 Memory Configuration

There are two UNIBUSes on the PDP-11/45 but in a single processor environment the second UNIBUS is generally connected into the first and becomes part of it. The existence of a second UNIBUS becomes significant where a high speed device would like to directly access the solid state memory. A device using the second UNIBUS must include a UNIBUS Priority Arbitration Unit, and the bus thus lends itself to multi-processor environments.

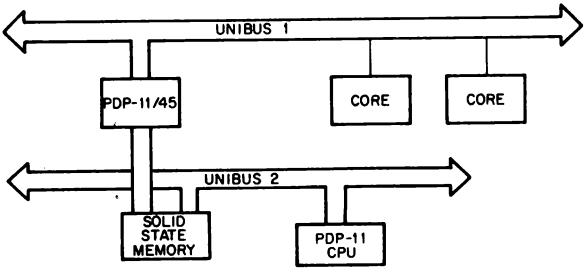


Figure 2-8 Multiprocessor Use of the Second UNIBUS

The UNIBUS and data path to the Solid State Memory are independent. While the Central Processor is operating on data in one Solid State Memory controller through the direct data path, any device could be using the UNIBUS to transfer information to core, to another device, or to the other Solid State Memory Controller. This autonomy significantly increases the throughput of the system.

2.5.2 Memory Retention

MOS memory bits have a capacitance which is charged to denote a 1 and uncharged to denote a 0. The entire MOS memory must be refreshed periodically, or the data will be lost. On the PDP-11/45, 1/32nd of the memory is refreshed every 60 microseconds. This process consumes only one solid state memory cycle.

The power required to refresh MOS memory is significantly less than that required for operation of the memory. Bipolar memory, on the other hand, does not require a refresh cycle but does require the same power to retain information as to operate.

2.5.3 Core Memory

The Central Processor communicates with core memory through the UNIBUS.

Each memory bank operates independently from other banks through its own controller which interfaces directly to the UNIBUS. Core memory can be continuously attached to the UNIBUS until the system contains a total of 124K (126,976 words) of memory.

An external device may use the UNIBUS to read or write core memory completely independently of, and simultaneously with the Central Processor's access of solid state memory. Furthermore, core memory and solid state memory may be used by the processor interchangeably.

2.5.4 Memory Interleaving

Generally, memory locations are numbered consecutively in a memory bank. Thus, when the address register is incremented on successive memory cycles the same bank is addressed and a full memory cycle must be completed before the new address can be used. The maximum data transfer rate for a devices using memory is therefore limited by memory cycle time. Memory interleaving is a technique that places consecutive word addresses in different memory banks and thus allows the write cycle in one memory to be overlapped with the read cycle in another.

The PDP-11/45's architecture with a controller inherent to each memory bank lends itself to memory interleaving, and mixing memories with different cycle times. The standard core memory has a basic cycle time of 850 nsec and is interleaved in pairs of 8,192 word units. A 16K system would be fully interleaved, whereas a 24K system (3 controllers) would only have 16K interleaved. Interleaving this memory provides the user with an effective cycle time of 650 nsec for consecutive accesses to sequential word locations.

The MOS memories are interleaved on an equal number of MOS blocks on each controller. Bipolar is not interleaved.

Memory interleaving is completely transparent to the computer programmer.

2.5.5 Mixing Memory

The PDP-11/45 can be used with the memory mix best suited to a user's needs. He can not only mix high speed solid state memories with magnetic core memories, but he can also choose core memories of different speeds. This is possible because of the independent nature of the core memory controllers.

The PDP 11/45 provides the user with an additional degree of freedom in mixing memories. The programmer need not address all of his solid state memory consecutively, but can intermix solid state and core physical addresses. Each solid state memory can address a 16K consecutive segment (32K when MOS is interleaved) beginning on a 16K boundary. If the controller contains a full 16K complement of MOS then the MOS will use up the full 16K address space; however, if there is less than 16K of MOS on the controller the user can intermix 4K blocks of core in with the 4K blocks of MOS.

When a program is using Memory Segmentation (Chapter 6) this manipulation of physical addresses is unnecessary as it may be done in the mapping of virtual space into physical space.

The user can reduce the cost of his system by buying only as much high speed memory as required; and he can increase system performance through the independence of data transfers on the UNIBUS and the CPU connection to the Solid State Memories.

2.5.6 Memory Parity

Memory Parity is optional for core and solid state memory. Parity words are extended to 18 bits and the last two bits (17,18) contain the parity indicators for the two bytes. Parity is generated when a word is written and checked when the word is read. Parity errors cause the Central Processor either to trap through location 4 or to halt. The user is referred to Appendix E for more information on memory parity.

2.5.7 Memory Organization

PDP-11 memories and instructions are designed to handle both 16-bit words, and 8-bit bytes. Therefore, since every word contains two bytes, a 4,096 word block contains 8,192 byte locations. Consecutive words are therefore found in even numbered addresses.

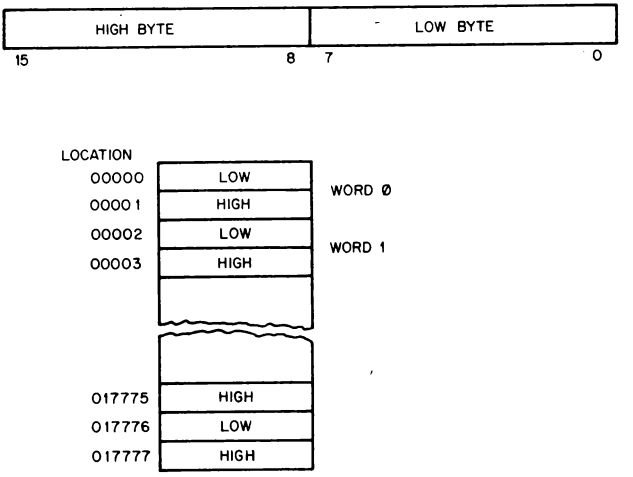


Figure 2-9 Memory Organization

Certain memory locations have been reserved by the system for interrupt and trap handling, and processor stacks, general registers, and peripheral device registers. Kernel virtual addresses from 0 to 370_8 are always reserved and those to 777_8 are reserved on large system configurations for traps and interrupt handling. The top 4,096 word addresses (from 770000_8 up) have been reserved for general registers and peripheral devices. Appendix C presents a detailed memory address map.

A 16-bit word used for byte addressing can address a maximum of 32K words. However, the top 4,096 word locations are traditionally reserved for peripheral and register addresses and the user therefore has 28K of core to program. To expand above 28K the user must use the Memory Segmentation Unit. This device, described in detail in Chapter 6, provides the programmer with an 18-bit effective memory address which permits him to address up to 126,976 words of actual memory. The unit also provides a memory management facility which permits individual user programs up to 64K in length (32K of instructions, 32K of data) and provides a relocation and protection facility through three sets of 16 registers.

2.6 SYSTEM INTERACTION

System intercommunication is carried out through the UNIBUS.

A device will request the UNIBUS for one of two purposes:

To make a non-processor (NPR) transfer of data. (Direct Data Transfers such as DMA), or

To interrupt program execution and force the processor to branch to a service routine.

There are two sources of interrupts, hardware and software.

2.6.1 Hardware Interrupt Requests

A hardware interrupt occurs when a device wishes to indicate to the program, or Central Processor, that a condition has occurred (such as transfer completed, end of tape, etc.). The interrupt can occur on any one of the four Bus Request levels and the processor responds to the interrupt through a service routine.

2.6.2 Program Interrupt Requests

Hardware interrupt servicing is often a two-level process. The first level is directly associated with the device's hardware interrupt and consists of retrieving the data. The second, is a software task that manipulates the raw information. The second process can be run at a lower priority than the first, because the PDP-11/45 provides the user with the means of scheduling his software servicing through seven levels of Program Interrupt Requests. The Program Interrupt Request Register is located at address 777772. An interrupt is generated by the programmer setting a bit in the high order byte of this register.

The reader is referred to Appendix D for more detailed information.

2.6.3 Priority Structure

When a device capable of becoming bus master requests use of the bus, handling of the request depends on the hierarchial positon of that device in the the priority structure.

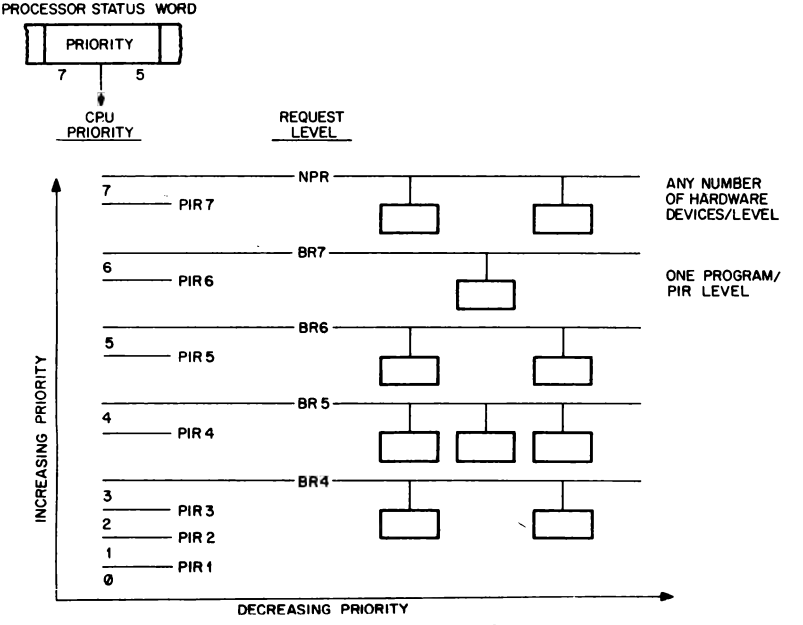


Figure 2-10 UNIBUS Priority Structure

The relative priority of the request is determined by the Processor's priority and the level at which the request is made.

The processor's priority is set under program control to one of eight levels using bits 7-5 in the processor Status Word. Bus requests are inhibited on the same or lower levels.

Bus requests from external devices can be made on any one of the five request lines. A non-processor request (NPR) has the highest priority, and its request is granted between bus cycles of an instruction execution. Bus Request 7 (BR 7) is the next highest priority and Bus Request 4 (BR 4) is the lowest. The four lower priority level requests (BR 7-BR 4) are granted by the processor between instructions providing that they occur on higher levels than the processor's, Therefore an interrupt may only occur on a Bus Request Level and not on a Non Processor Request level.

Any number of devices can be connected to a specific BR or NPR line.

If two devices with the same priority request the bus, the device physically closest to the processor on the UNIBUS has the higher priority.

Program Interrupt Requests can be made on any one of 7 levels (PIR 7-PIR 1). Requests are granted by the processor between instructions providing that they occur on higher levels than the processor's.

Program Interrupt Requests take precedence over equivalent level Bus Requests.

2.6.4 Non-Processor Data Transfers

Direct memory or direct data transfers can be accomplished between any two peripherals without processor supervision. These Non-Processor transfers, called NPR level data transfers, are usually made for Direct Memory Access (memory to/from mass storage) or direct device transfers (disk refreshing a CRT display).

A NPR device provides extremely fast access to the UNIBUS and can transfer data at high rates once it gains control of the bus. The state of the processor is not affected by this type of transfer, and, therefore, the processor can relinquish bus control while an instruction is still in progress. The bus can be released at the end of any bus cycle, except during a read-modify-write cycle sequence. (This occurs for example in destructive read-out devices such as core memory for certain instructions).

In the PDP-11/45 an NPR device can gain bus control in 3.5 microseconds or less (depending on the number of devices on the UNIBUS), and can transfer 16-bit words to memory at the same speed as the effective cycle time of the memory being addressed.

2.6.5 Using the Interrupts

Devices that gain bus control with one of the Bus Request lines (BR 7 - BR 4), can take full advantage of the Central Processor by requesting an interrupt. In this way, the entire instruction set is available for manipulating data and status registers.

When a service routine is to be run, the current task being performed by the central processor is interrupted, and the device service routine is initiated. Once the request has been satisfied, the Processor returns to its former task. Interrupts may also be used to schedule program execution by using the Program Interrupt Request.

2.6.6 Interrupt Procedure

Interrupt handling is automatic in the PDP-11/45. No device polling is required to determine which service routine to execute. The operations required to service an interrupt are as follows:

1. Processor relinquishes control of the bus, priorities permitting.
2. When a master gains control, it sends the processor an interrupt command and an unique memory address which contains the address of the device's service routine in Kernel virtual address space, called the interrupt vector address. Immediately following this pointer address is a word (located at vector address +2) which is to be used as a new Processor Status Word.
3. The processor stores the current Processor Status Word (PS) and the current Program Counter (PC) into CPU temporary registers.
4. The new PC and PS (the interrupt vector) are taken from the specified address. The old PS and PC are then pushed onto the current stack as indicated by bits 15,14 of the new PS and the previous mode in effect is stored in bits 13,12 of the new PS. The service routine is then initiated.

These operations are performed in 2.5 μsec from the time the control processor receives the interrupt command until the time it starts executing the first instruction of the service routine. This time interval assumes no NPR transfers occured during this time interval.

5. The device service routine can cause the processor to resume the interrupted process by executing the Return from Interrupt (RTI or RTT) instruction, described in Chapter 4, which pops the two top words from the current processor stack and uses them to load the PC and PS registers.

This instruction requires 1.5 μsec providing there is no NPR request.

A device routine can be interrupted by a higher priority bus request any time after the new PC and PS have been loaded. If such an interrupt occurs, the PC and the PS of the service routine are automatically stored in the temporary registers and then pushed onto the new current stack, and the new device routine is initiated.

2.6.7 Interrupt Servicing

Every hardware device capable of interrupting the processor has a unique set of locations (2 words) reserved for its interrupt vector. The first word contains the location of the device's service routine, and the second, the Processor Status Word that is to be used by the service routine. Through proper use of the PS, the programmer can switch the operational mode of the processor, alter the General Register Set in use (context switching), and modify the Processor's Priority level to mask out lower level interrupts.

There is one interrupt vector for the Program Interrupt Request. It will generally be necessary in a multi-processing environment to determine which program generated the PIR and where it is located in memory. Appendix D provides an example of how this is done.

2.7 PROCESSOR TRAPS

There are a series of errors and programming conditions which will cause the Central Processor to trap to a set of fixed locations. These include Power Failure, Odd Addressing Errors, Stack Errors, Timeout Errors, Memory Parity Errors, Segmentation Violations, Floating Point Processor Exception Traps, Use of Reserved Instructions, Use of the T bit in the Processor Status Word, and use of the IOT, EMT, and TRAP instructions.

Stack Errors, Memory Parity Errors, and the T bit Trap have already been discussed in this chapter. Segmentation Violations and Floating Point Exception Traps are described in Chapters 6 and 7 respectively. The IOT, EMT, and TRAP instructions are described in Chapter 4.

2.7.1 Power Failure

Whenever AC power drops below 95 volts for 110v power (190 volts for 220v) or outside a limit of 47 to 63 HZ, as measured by DC power, the power fail sequence is initiated. The Central Processor automatically traps to location 24 and the power fail program has 2 msec. to save all volatile information (data in registers), and condition peripherals for power fail.

When power is restored the processor traps to location 24 and executes the power up routine to restore the machine to its state prior to power failure.

2.7.2 Odd Addressing Errors

This error occurs whenever a progam attempts to execute a word instruction on an odd address (in the middle of a word boundary). The instruction is aborted and the CPU traps through location 4.

2.7.3 Time-out Errors

These errors occur when a Master Synchronization pulse is placed on the UNIBUS and there is no slave pulse within 5 μsec. This error usually occurs in attempts to address non-existant memory or peripherals.

The offending instruction is aborted and the processor traps through location 4.

2.7.4 Reserved Instructions

There is a set of illegal and reserved instructions which cause the processor to trap through Location10. The set is fully described in Appendix C.

2.7.5 Trap Handling

Appendix C includes a list of the reserved Trap Vector locations, and System Error Definitions which cause processor traps. When a trap occurs, the processor follows the same procedure for traps as it does for interrupts (saving the PC and PS on the new Processor Stack etc...)

In cases where traps and interrupts occur concurrently, the processor will service the conditions according to the priority sequence illustrated in Figure 2-11.

Odd Addressing Error

Fatal Stack Violations (Red)

Memory Segmentation Violations

Timeout Errors

Parity Errors

Floating Point Processor Transfer Request

Memory Segment Management Traps

Warning Stack Violation (Yellow)

Power Failure

Processor Priority level 7

Floating Point Exception Trap

PIR 7

BR 7

PIR 1

Processor 0

Figure 2-11 Processor Service Hierarchy

Appendix C includes more details on the Trap sequence and Trap/Interrupt interaction.

2.8 MULTI-PROGRAMMING

The PDP 11/45's architecture with its three modes of operation, its two sets of general registers, its Memory Segmentation and its Program Interrupt Request facility provides an ideal environment for multi-programming systems.

In any multi-programming system there must be some method of transferring information and control between programs operating in the same or different modes. The PDP 11/45 provides the user with these communication paths.

2.8.1 Control Information

Control is passed inwards (User, Supervisor, Kernel) by all traps and interrupts. All trap and interrupt vectors are located in Kernel virtual space. Thus all traps and interrupts pass through Kernel space to pick up their new PC and PS and determine the new mode of processing.

Control is passed outwards (Kernel, Supervisor, User) by the RTI and RTT instructions (described in Chapter 4).

2.8.2 Data

Data is transferred between modes by four instructions Move From Previous Instruction space (MFPI), Move From Previous Data space (MFPD), Move To Previous Instruction space (MTPI) and Move To Previous Data space (MTPD). There are four instructions rather than two as Memory Segmentation distinguishes between instructions and data (Chapter 6). The instructions are fully described in Chapter 4. However, it should be noted that these instructions have been designed to allow data transfers to be under the control of the innermost mode (Kernel, Supervisor, User) program and not the outermost, thus providing protection of an inner program from an outer.*

2.8.3 Processor Status Word

The PDP 11/45 protects the PS from implicit references by Supervisor and User programs which could result in damage to an inner level program.

A program operating in Kernel mode can perform any manipulation of the PS. Programs operating at outer levels (Supervisor and User) are inhibited from changing bits 5-7 (the Processor's Priority). They are also restricted in their treatment of bits 15, 14 (Current Mode), bits 13, 12 (Previous Mode), and bit 11 (Register Set); these bits may only be set, they are only cleared by an interrupt or trap.

Thus, a programmer can pass control outwards through the RTI and RTT instructions to set bits in the mode fields of his PS. To move inwards, however, bits must be cleared and he must therefore, issue a trap or interrupt.

The Kernel can further protect the PS from explicit references (Move data to location 777776-the PS) through Memory Segmentation.

CHAPTER 3

ADDRESSING MODES

Data stored in memory must be accessed, and manipulated. Data handling is specified by a PDP-11 instruction (MOV, ADD etc.) which usually indicates:

the function (operation code)

a general purpose register to be used when locating the source operand and/or a general purpose register to be used when locating the destination operand.

an addressing mode (to specify how the selected register(s) is/are to be used)

Since a large portion of the data handled by a computer is usually structured (in character strings, in arrays, in lists etc.), the PDP-11 has been designed to handle structured data efficiently and flexibly. The general registers may be used with an instruction in any of the following ways:

as accumulators. The data to be manipulated resides within the register.

as pointers. The contents of the register are the address of the operand, rather than the operand itself.

as pointers which automatically step through core locations. Automatically stepping forward through consecutive core locations is known as autoincrement addressing; automatically stepping backwards is known as autodecrement addressing. These modes are particularly useful for processing tabular data.

as index registers. In this instance the contents of the register, and the word following the instruction are summed to produce the address of the operand. This allows easy access to variable entries in a list.

PDP-11's also have instruction addressing mode combinations which facilitate temporary data storage structures for convenient handling of data which must be frequently accessed. This is known as the " stack." (see Chapter 5)

In the PDP-11 any register can be used as a "stack pointer"under program control, however, certain instructions associated with subroutine linkage and interrupt service automatically use Register 6 as a "hardware stack pointer". For this reason R6 is frequently referred to as the "SP".

R7 is used by the processor as its program counter (PC). It is recommended that R7 not be used as a stack pointer.

An important PDP-11/45 feature, which must be considered in conjunction with the addressing modes, is the register arrangement;

Two sets of general purpose registers (R0-R5)

three hardware stack pointers (R6)

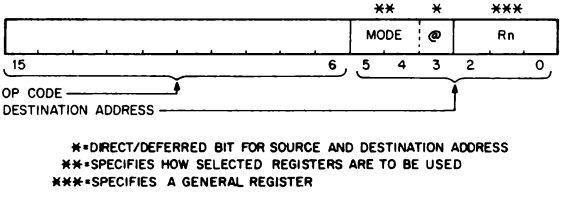
a single Program Counter (PC) register (R7).

Register R7 is used as a common program counter (PC). At any point in time only one register set is active. Thus a programmer need only concern himself with the existence of multiple register sets for those special supervisory tasks which involve Kernel, Supervisor, User communications (e.g. MTPX, MFPX); otherwise he need never worry about which R3 or R6 an instruction will reference, the choice is automatic and transparent to his program.

Instruction mnemonics and address mode symbols are sufficient for writing machine language programs. The programmer need not be concerned about conversion to binary digits; this is accomplished automatically by the PDP-11/45 assembler.

3.1 SINGLE OPERAND ADDRESSING

The instruction format for all single operand instructions (such as clear, increment, test) is:



Bits 15 through 6 specify the operation code that defines the type of instruction to be executed.

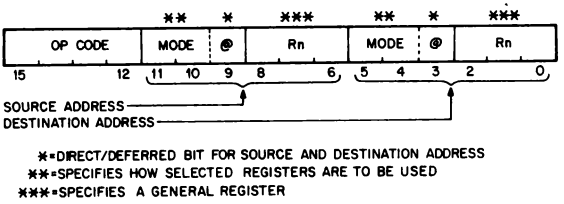
Bits 5 through 0 form a six-bit field called the destination address field. This consists of two subfields:

a) Bits 0 through 2 specify which of the eight general purpose registers is to be referenced by this instruction word.

b) Bits 4 and 5 specify how the selected register will be used (address mode). Bit 3 indicates direct or deferred (indirect) addressing.

3.2 DOUBLE OPERAND ADDRESSING

Operations which imply two operands (such as add, subtract, move and compare) are handled by instructions that specify two addresses. The first operand is called the source operand, the second the destination operand. Bit assignments in the source and destination address fields may specify different modes and different registers. The Instruction format for the double operand instruction is:



The source address field is used to select the source operand, the first operand. The destination is used similarly, and locates the second operand and the result. For example, the instruction ADD A,B adds the contents (source operand) of location A to the contents (destination operand) of location B. After execution B will contain the result of the addition and the contents of A will be unchanged.

Examples in this section and further in this chapter use the following sample PDP-11 instructions:

Mnemonic	Description	Octal Code
CLR	clear (zero the specified destination)	0050nn
CLRB	clear byte (zero the byte in the specified destination)	1050nn
INC	increment (add 1 to contents of destination)	0052nn
INCB	increment byte (add 1 to the contents of destination byte)	1052nn
COM	complement (replace the contents of the destination by their logical complement; each 0 bit is set and each 1 bit is cleared)	0051nn
COMB	complement byte (replace the contents of the destination byte by their logical complement; each 0 bit is set and each 1 bit is cleared).	1051nn
ADD	add (add source operand to destination operand and store the result at destination address)	06mmnn

3.3 DIRECT ADDRESSING

The following table summarizes the four basic modes used with direct addressing.

DIRECT MODES

Binary	Name	Assembler Syntax	Function
0 0 0	Register	Rn	Register contains operand
0 1 0	Autoincrement	(Rn)+	Register is used as a pointer to sequential data then incremented
1 0 0	Autodecrement	-(Rn)	Register is decremented and then used as a pointer.
1 1 0	Index	X(Rn)	Value X is added to (Rn) to produce address of operand. Neither X nor (Rn) are modified.

3.3.1 Register Mode

OPR Rn

With register mode any of the general registers may be used as simple accumulators and the operand is contained in the selected register. Since they are hardware registers, within the processor, the general registers operate at high speeds and provide speed advantages when used for operating on frequently-accessed variables. The PDP-11 assembler interprets and assembles instructions of the form OPR Rn as register mode operations. Rn represents a general register name or number and OPR is used to represent a general instruction mnemonic. Assembler syntax requires that a general register be defined as follows:

R0 = %0 (% sign indicates register definition)

R1 = %1

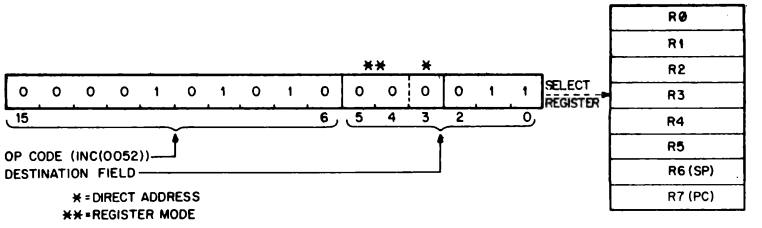
R2 = %2, etc.

Registers are typically referred to by name as R0, R1, R2, R3, R4, R5, R6 and R7. However R6 and R7 are also referred to as SP and PC, respectively.

Register Mode Examples
(all numbers in octal)

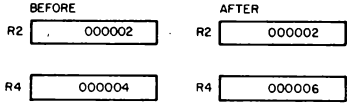
	Symbolic	Octal Code	Instruction Name
1.	INC R3	005203	Increment

Operation: Add one to the contents of general register 3



2. ADD R2,R4 060204 Add

Operation: Add the contents of R2 to the contents of R4.



3. COMB R4 105104 Complement Byte

Operation: One's complement bits 0-7 (byte) in R4. (When general registers are used, byte instructions only operate on bits 0-7; i.e. byte 0 of the register)

3.3.2 Autoincrement Mode

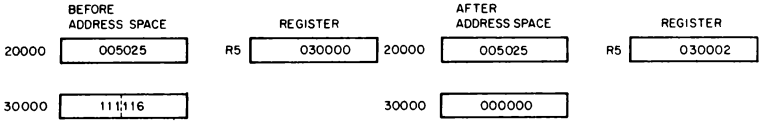
OPR (Rn)+

This mode provides for automatic stepping of a pointer through sequential elements of a table of operands. It assumes the contents of the selected general register to be the address of the operand. Contents of registers are stepped (by one for bytes, by two for words, always by two for R6 and R7) to address the next sequential location. The autoincrement mode is especially useful for array processing and stacks. It will access an element of a table and then step the pointer to address the next operand in the table. Although most useful for table handling, this mode is completely general and may be used for a variety of purposes.

Autoincrement Mode Examples

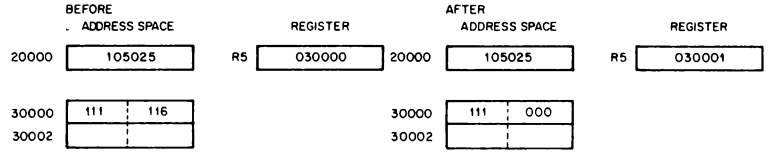
	Symbolic	Octal Code	Instruction Name
1.	CLR (R5) +	005025	Clear

Operation: Use contents of R5 as the address of the operand. Clear selected operand and then increment the contents of R5 by two.



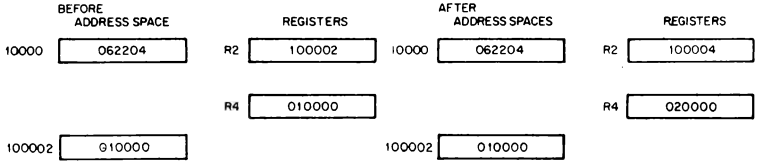
2.	CLRB (R5) +	105025	Clear Byte

Operation: Use contents of R5 as the address of the operand. Clear selected byte operand and then increment the contents of R5 by one.



3.	ADD (R2) +,R4	062204	Add

Operation: The contents of R2 are used as the address of the operand which is added to the contents of R4. R2 is then incremented by two.



3.3.3 Autodecrement Mode

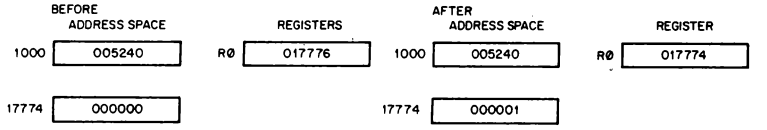
OPR-(Rn)

This mode is useful for processing data in a list in reverse direction. The contents of the selected general register are decremented (by two for word instructions, by one for byte instructions) and then used as the address of the operand. The choice of postincrement, predecrement features for the PDP-11 were not arbitrary decisions, but were intended to facilitate hardware/software stack operations (See Chapter 5 for complete discussions of stacks).

Autodecrement Mode Examples

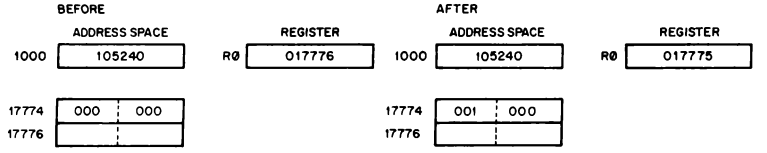
	Symbolic	Octal Code	Instruction Name
1.	INC-(R0)	005240	Increment

Operation: The contents of R0 are decremented by two and used as the address of the operand. The operand is increased by one.



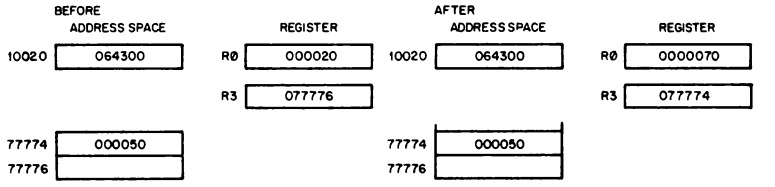
2.	INCB-(R0)	105240	Increment Byte

Operation: The contents of R0 are decremented by one then used as the address of the operand. The operand byte is increased by one.



3.	ADD-(R3),R0	064300	Add

Operation: The contents of R3 are decremented by 2 then used as a pointer to an operand (source) which is added to the contents of R0 (destination operand).



3.3.4 Index Mode

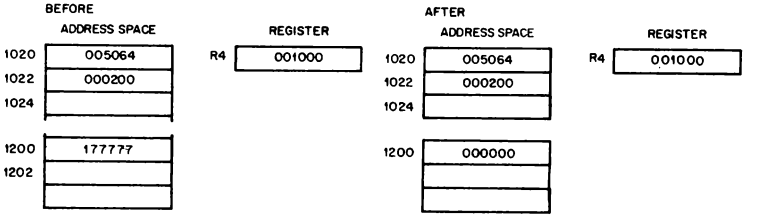
OPR X(Rn)

The contents of the selected general register, and an index word following the instruction word, are summed to form the address of the operand. The contents of the selected register may be used as a base for calculating a series of addresses, thus allowing random access to elements of data structures. The selected register can then be modified by program to access data in the table. Index addressing instructions are of the form OPR X(Rn) where X is the indexed word and is located in the memory location following the instruction word and Rn is the selected general register.

Index Mode Examples

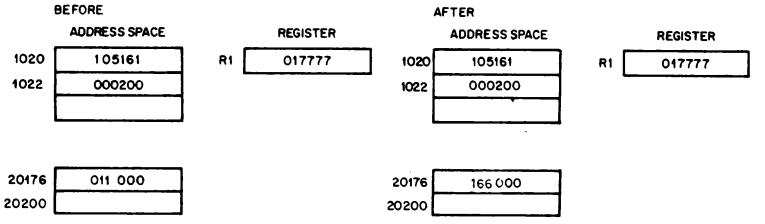
	Symbolic	Octal Code	Instruction Name
1.	CLR 200(R4)	005064 000200	Clear

Operation: The address of the operand is determined by adding 200 to the contents of R4. The location is then cleared.



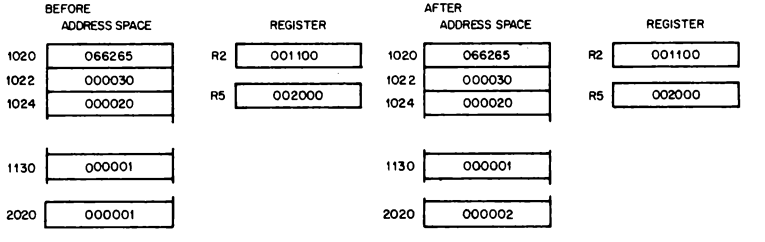
2.	COMB 200(R1)	105161 000200	Complement Byte

Operation: The contents of a location which is determined by adding 200 to the contents of R1 are one's complemented. (i.e. logically complemented)



3. ADD 30(R2),20(R5) 066265 Add
000030
000020

Operation: The contents of a location which is determined by adding 30 to the contents of R2 are added to the contents of a location which is determined by adding 20 to the contents of R5. The result is stored at the destination address, i.e. 20(R5)



3.4 DEFERRED (INDIRECT) ADDRESSING

The four basic modes may also be used with deferred addressing. Whereas in the register mode the operand is the contents of the selected register, in the register deferred mode the contents of the selected register is the address of the operand.

In the three other deferred modes, the contents of the register selects the address of the operand rather than the operand itself. These modes are therefore used when a table consists of addresses rather than operands. Assembler syntax for indicating deferred addressing is "@" (or "( )" when this not ambiguous). The following table summarizes the deferred versions of the basic modes:

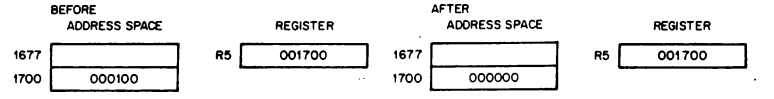
Binary Code	Name	Assembler Syntax	Function
0 0 1	Register Deferred	@Rn or (Rn)	Register contains the address of the operand
0 1 1	Autoincrement Deferred	@(Rn)+	Register is first used as a pointer to a word containing the address of the operand, then incremented (always by 2; even for byte instructions).
1 0 1	Autodecrement Deferred	@-(Rn)	Register is decremented (always by two; even for byte instructions) and then used as a pointer to a word containing the address of the operand
1 1 1	Index Deferred	@X(Rn)	Value X (stored in a word following the instruction) and (Rn) are added and the sum is used as a pointer to a word containing the address of the operand. Neither X nor (Rn) are modified.

Since each deferred mode is similar to its basic mode counterpart, separate descriptions of each deferred mode are not necessary. However, the following examples illustrate the deferred modes.

Register Deferred Mode Example

Symbolic	Octal Code	Instruction Name
CLR @R5	005015	Clear

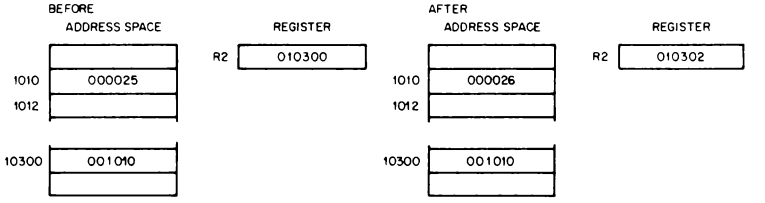
Operation: The contents of location specified in R5 are cleared.



Autoincrement Deferred Mode Example

Symbolic	Octal Code	Instruction Name
INC@(R2)+	005232	Increment

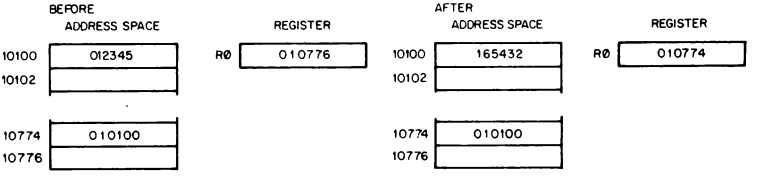
Operation: The contents of R2 are used as the address of the address of the operand.
Operand is increased by one. Contents of R2 is incremented by 2.



Autodecrement Deferred Mode Example

Symbolic	Octal Code	Complement
COM @-(R0)	005150	

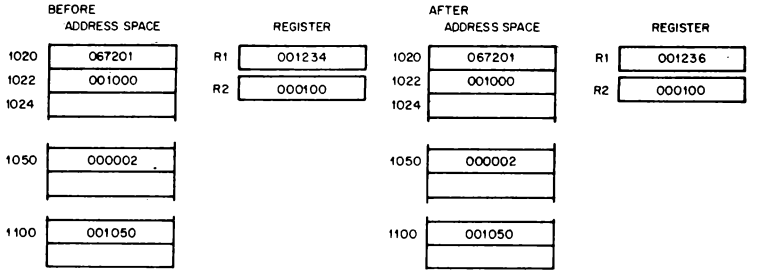
Operation: The contents of R0 are decremented by two and then used as the address of the address of the operand. Operand is one's complemented. (i.e. logically complemented)



Index Deferred Mode Example

Symbolic	Octal Code	Instruction Name
ADD @ 1000(R2),R1	067201 001000	Add

Operation: 1000 and contents of R2 are summed to produce the address of the address of the source operand the contents of which are added to contents of R1; the result is stored in R1.



3.5 USE OF THE PC AS A GENERAL REGISTER

Although Register 7 is a general purpose register, it doubles in function as the Program Counter for the PDP-11. Whenever the processor uses the program counter to acquire a word from memory, the program counter is automatically incremented by two to contain the address of the next word of the instruction being executed or the address of the next instruction to be executed. (When the program uses the PC to locate byte data, the PC is still incremented by two.)

The PC responds to all the standard PDP-11 addressing modes. However, there are four of these modes with which the PC can provide advantages for handling position independent code (PIC - seeChapter 5) and unstructured data. When regarding the PC these modes are termed immediate, absolute (or immediate deferred), relative and relative deferred, and are summarized below:

Binary Code	Name	Assembler Syntax	Function
0 1 0	Immediate	#n	Operand follows instruction
0 1 1	Absolute	@#A	Absolute Address folows instruction
1 1 0	Relative	A	Address of A, relative to the instruction, follows the instruction.
1 1 1	Relative Deferred	@A	Address of location containing address of A, relative to the instruction follows the instruction.

The reader should remember that the special effect modes are the same as modes described in 3.3 and 3.4, but the general register selected is R7, the program counter.

When a standard program is available for different users, it often is helpful to be able to load it into different areas of core and run it there. PDP-11's can accomplish the relocation of a program very efficiently through the use of position inde-

pendent code (PIC) which is written by using the PC addressing modes. If an instruction and its objects are moved in such a way that the relative distance between them is not altered, the same offset relative to the PC can be used in all positions in memory. Thus, PIC usually references locations relative to the current location. PIC is discussed in more detail in Chapter 5.

The PC also greatly facilitates the handling of unstructured data. This is particularly true of the immediate and relative modes which are discussed more fully in Paragraphs 3.5.1 and 3.5.2.

3.5.1 Immediate Mode

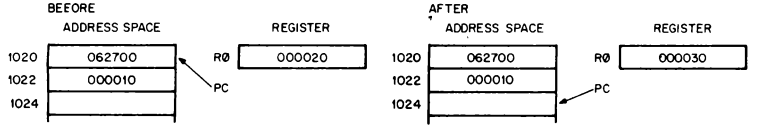
OPR #n,DD

Immediate mode is equivalent to using the autoincrement mode with the PC. It provides time improvements for accessing constant operands by including the constant in the memory location immediately following the instruction word.

Immediate Mode Example

Symbolic	Octal Code	Instruction Name
ADD #10,R0	062700 000010	Add

Operation: The value 10 is located in the second word of the instruction and is added to the contents of R0. Just before this instruction is fetched and executed, the PC points to the first word of the instruction. The processor fetches the first word and increments the PC by two. The source operand mode is 27 (autoincrement the PC). Thus, the PC is used as a pointer to fetch the operand (the second word of the instruction) before being incremented by two to point to the next instruction.



3.5.2 Absolute Addressing

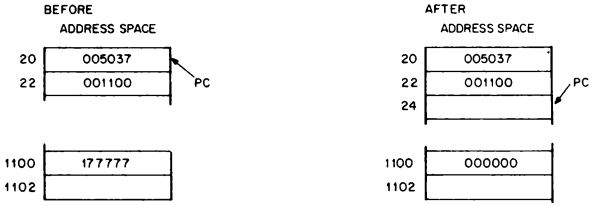
OPR @#A

This mode is the equivalent of immediate deferred or autoincrement deferred using the PC. The contents of the location following the instruction are taken as the address of the operand. Immediate data is interpreted as an absolute address (i.e., an address that remains constant no matter where in memory the assembled instruction is executed).

Absolute Mode Examples

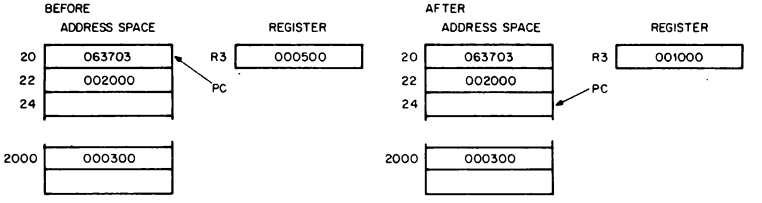
	Symbolic	Octal Code	Instruction Name
1.	CLR @#1100	005037 001100	Clear

Operation: Clear the contents of location 1100.



2.	ADD @#2000,R3	063703 002000	

Operation: Add contents of location 2000 to R3.



3.5.3 Relative Addressing

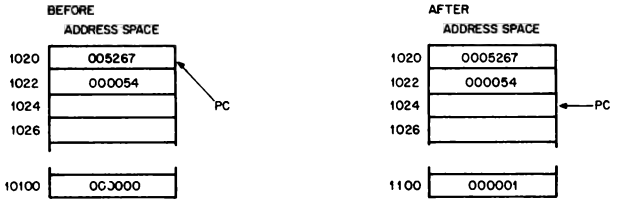
OPR A or
OPR X(PC) , where X is the location of A relative to the instruction.

This mode is assembled as index mode using R7. The base of the address calculation, which is stored in the second or third word of the instruction, is not the address of the operand, but the number which, when added to the (PC), becomes the address of the operand. This mode is useful for writing position independent code (see Chapter 5) since the location referenced is always fixed relative to the PC. When instructions are to be relocated, the operand is moved by the same amount.

Relative Addressing Example

Symbolic	Octal Code	Instruction Name
INC A	005267 000054	Increment

Operation: To increment location A, contents of memory location immediately following instruction word are added to (PC) to produce address A. Contents of A are increased by one.



3.5.4 Relative Deferred Addressing

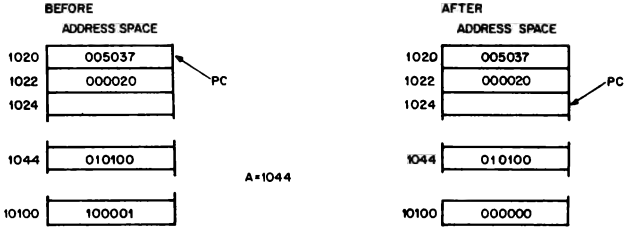
OPR@A or OPR@X(PC), where x is location containing address of A, relative to the instruction.

This mode is similar to the relative mode, except that the second word of the instruction, when added to the PC, contains the address of the address of the operand, rather that the address of the operand.

Relative Deferred Mode Example

Symbolic	Octal Code	Instruction Name
CLR @A	005077 000020	Clear

Operation: Add second word of instruction to PC to produce address of address of operand. Clear operand.



3.6 USE OF STACK POINTER AS GENERAL REGISTER

The processor stack pointer (SP, Register 6) is in most cases the general register used for the stack operations related to program nesting. Autodecrement with Register 6 "pushes"data on to the stack and autoincrement with Register 6 "pops" data off the stack. Index mode with the SP permits random access of items on the stack. Since the SP is used by the processor for interrupt handling, it has a special attribute: autoincrements and autodecrements are always done in steps of two. Byte operations using the SP in this way simply leave odd addresses unmodified. Use of stacks is explained in detail in Chapter 5.

On the PDP-11/45 there are three R6 registers selected by the PS; but at any given time there is only one in operation.

The following table is a concise summary of the various PDP-11 addressing modes

DIRECT MODES

Binary Code	Name	Assembler Syntax	Function
000	Register	Rn	Register contains operand
010	Autoincrement	(Rn)+	Register contains address of operand. Register contents incremented after reference.
100	Autodecrement	-(Rn)	Register contents decremented before reference register contains address of operand
110	Index	X(Rn)	Value X (stored in a word following the instruction) is added to (Rn) to produce address of operand. Neither X nor (Rn) are modified.

DEFERRED MODES

Binary Code	Name	Assembler Syntax	Function
001	Register Deferred	@Rn or (Rn)	Register contains the address of the operand
011	Autoincrement Deferred	@(Rn)+	Register is first used as a pointer to A word containing the address of the operand, then incremented (always by 2; even for byte instructions)
101	Autodecrement Deferred	@-(Rn)	Register is decremented (always by two; even for byte instructions) and then used as a pointer to a word containing the address of the operand
111	Index Deferred	@X(Rn)	Value X (stored in a word following the instruction) and (Rn) are added and the sum is used as a pointer to a word containing the address of the operand. Neither X nor (Rn) are modified

PC ADDRESSING

010	Immediate	#n	Operand follows instruction
011	Absolute	@#A	Absolute address follows instruction
110	Relative	A	Address of A, relative to the instruction, follows the instruction.
111	Relative Deferred	-@A	Address of location containing address of A, relative to the instruction follows the instruction.

CHAPTER 4

INSTRUCTION SET

4.1 INTRODUCTION

This chapter describes the PDP-11/45 instructions in the following order:

Single Operand (4.4)

General

Shifts

Multiple Precision

Rotates

Double Operand (4.5)

Arithmetic Instructions

General Register Destination

Logical Instructions

Program Control Instructions (4.6)

Branches

Subroutines

Traps

Miscellaneous (4.7)

Condition Code Operators (4.8)

The specification for each instruction includes the mnemonic, octal code, binary code, a diagram showing the format of the instruction, a symbolic notation describing its execution and the effect on the condition codes, timing information, a description, special comments, and examples.

MNEMONIC: This is indicated at the top corner of each page. When the word instruction has a byte equivalent, the byte mnemonic is also shown.

INSTRUCTION FORMAT: A diagram accompanying each instruction shows the octal op code, the binary op code, and bit assignments. (Note that in byte instructions the most significant bit (bit 15) is always a 1.)

OPERATION: The operation of each instruction is described with a single notation. The following symbols are used:

( ) = contents of

src = source address

dst = destination address

loc = location

← = becomes

↑ = "is popped from stack"

↓ = "is pushed onto stack"

∧ = boolean AND

∨ = boolean OR

⊻ = exclusive OR

~ = boolean not

Reg or R = register

B = Byte

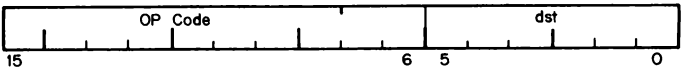
INSTRUCTION TIMING: The minimum execution time, including the fetch of the next instruction, is specified for each instruction. The instruction is assumed to reside in bipolar memory; both source and destination are assumed to be general purpose registers. For example, MOV is assumed to be from a general register to a general register, and JMP is assumed to be in register deferred mode. For detailed timing information consult Appendix B.

ISP - The Instruction Set Processor (ISP) notation has been used with each instruction. It is a precise notation for defining the action of any instruction set and is described in detail in Appendix A. It was included for the benefit of PDP-11 users who wish to gain an in depth understanding of each instruction. However, understanding ISP is not essential to understanding PDP-11 instructions.

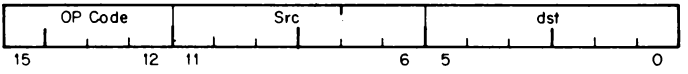
4.2 INSTRUCTION FORMATS

The major instruction formats are:

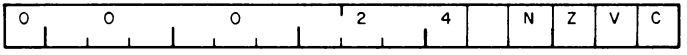
Single Operand Group



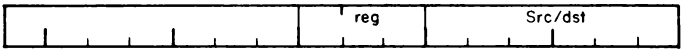
Double Operand Group



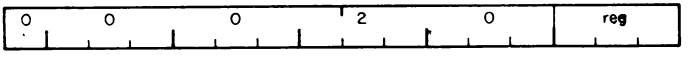
Condition Code Operators



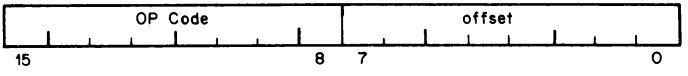
Register-Source or Destination



Subroutine Return

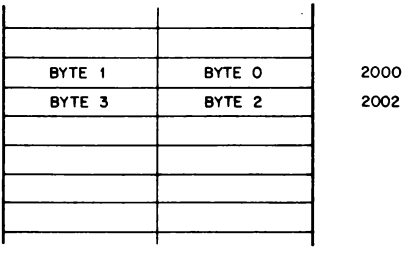


Branch



4.3 BYTE INSTRUCTIONS

The PDP-11 processor includes a full complement of instructions that manipulate byte operands. Since all PDP-11 addressing is byte-oriented, byte manipulation addressing is straightforward. Byte instructions with autoincrement or autodecrement direct addressing cause the specified register to be modified by one to point to the next byte of data. Byte operations in register mode access the low-order byte of the specified register. These provisions enable the PDP-11 to perform as either a word or byte processor. The numbering scheme for word and byte addresses in core memory is:



The most significant bit (Bit 15) of the instruction word is set to indicate a byte instruction.

Example:

	Symbolic	Octal
CLR		0050DD
CLRB		1050DD

4.4 SINGLE OPERAND INSTRUCTIONS

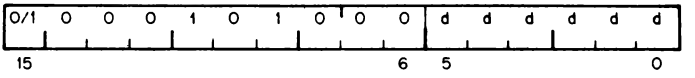
4.4.1 Single Operand Arithmetic Instructions

General:	CLR	DEC	INC	NEG	TST	COM
	CLRB	DECB	INCB	NEGB	TSTB	COMB
Shifts:	ASR	ASL	ASH	ASHC		
	ASRB	ASLB				
Multiple Precision:	ADC	SBC	SXT			
	ADCB	SBCB				
Rotates:	ROL	ROR	SWAB			
	ROLB	RORB				

300 ns

CLR
CLRB

Clear dst n050DD



Operation: (dst)←0

Condition Codes:
N: cleared
Z: set
V: cleared
C: cleared

Description: Word: Contents of specified destination are replaced with zeroes.
Byte: Same

Example:

CLR R1

Before	After
(R1) = 177777	(R1) = 000000
N Z V C	N Z V C
1 1 1 1	0 1 0 0

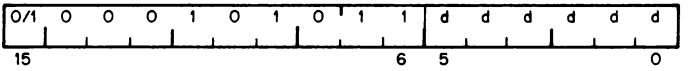
ISP:

```
CLR:
    D' ← 0;                          clear D, N, V, C, set Z
    N ← 0;
    Z ← 1;
    V ← 0;
    C ← 0

CLRB:
    Db' ← 0;                         clear D, N, V, C; set Z
    N ← 0;
    Z ← 1;
    V ← 0;
    C ← 0
```

DEC
DECB

Decrement dst

n053DD



Operation: (dst)←(dst)−1

Condition Codes:
N: set if result is <0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if (dst) was 100000; cleared otherwise
C: not affected

Description: Word: Subtract 1 from the contents of the destination
Byte: Same

Example:

DEC R5

Before	After
(R5) = 000001	(R5) = 000000
N Z V C	N Z V C
1 0 0 0	0 1 0 0

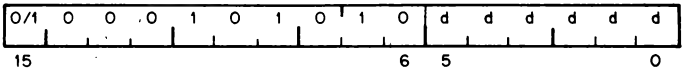
ISP:

```
DEC:
   r ← D'-1; next                                  result is difference of D-1
      N ← r<15>;                                   negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);           zero?
     (r<15:0> = 77777₈) ⇒ (V ← 1 else V ← 0);      overflow if largest positive number
      D ← r                                        transmit result to D


DECB:
   r ← Db'-1; next                                 result is difference of D-1
      N ← r<7>;                                    negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);            zero?
     (r<7:0> = 177₈) ⇒ (V ← 1 else V ← 0);         overflow if largest positive number
      Db ← r                                       transmit result to D
```

300 ns

INC
INCB

Increment dst n052DD



Operation: (dst)←(dst) + 1

Condition Codes:
N: set if result is <0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if (dst) held 077777; cleared otherwise
C: not affected

Description:
Word: Add one to contents of destination
Byte: Same

Example:

INC R2

Before	After
(R2) = 000333	(R2) = 000334
N Z V C	N Z V C
0 0 0 0	0 0 0 0

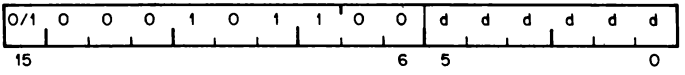
ISP:

```
INC:
   r ← D'+1; next                          result is sum of D+1
      N ← r<15>;                           negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);   zero?
     (r<15:0> = 100000₈) ⇒ (V ← 1 else V ← 0); overflow if largest negative number
      D ← r                                transmit result to D

INCB:
   r ← Db+1; next                          result is sum of D+1
      N ← r<7>;                            negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);    zero?
     (r<7:0> = 200₈) ⇒ (V ← 1 else V ← 0); overflow if largest negative number
      Db ← r                               transmit result to D
```

NEG
NEGB

Negate dst

n0054DD



Operation: (dst)← -(dst)

Condition Codes:
N: set if the result is <0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if the result is 100000; cleared otherwise
C: cleared if the result is 0; set otherwise

Description: Word: Replaces the contents of the destination address by its two's complement. Note that 100000 is replaced by itself -(in two's complement notation the most negative number has no positive counterpart).
Byte: Same

Example:

NEG R0

Before	After
(R0) = 000010	(R0) = 177770
N Z V C	N Z V C
0 0 0 0	1 0 0 1

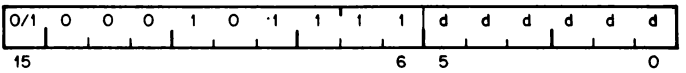
ISP:

```
NEG:
   r ← -D'; next                                  result is negative of D
     N ← r<15>;                                   negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);          zero?
     (r<15:0> = 100000₈) ⇒ (V ← 1 else V ← 0);    overflow?
     (r<15:0> = 0) ⇒ (C ← 0 else C ← 1);          carry?
      D ← r                                       transmit result to D

NEGB:
   r ← - Db'; next                                result is negative of D
     N ← r<7>;                                    negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);           zero?
     (r<7:0> = 200₈) ⇒ (V ← 1 else V ← 0);        overflow?
     (r<7:0> = 0) ⇒ (C ← 0 else C ← 1);           carry?
      Db ← r                                      transmit result to D
```

300 ns

TST
TSTB

Test dst n057DD



Operation: (dst)←(dst)

Condition Codes:
N: set if the result is <0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: cleared
C: cleared

Description: Word: Sets the condition codes N and Z according to the contents of the destination address
Byte: Same

Example:

TST R1

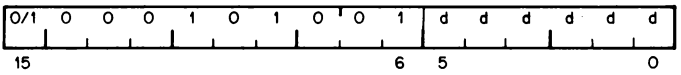
Before	After
(R1) = 012340	(R1) = 012340
N Z V C	N Z V C
0 0 1 1	0 0 0 0

ISP:

```
TST:
   r ← D' - 0; next                          result is difference of D and 0
      N ← r<15>;                             negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);     zero?
      V ← 0;                                 clear V and C
      C ← 0

TSTB:
   r ← Db' - 0; next                         result is difference of D and 0
      N ← r<7>;                              negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);      zero?
      V ← 0;                                 clear V and C
      C ← 0
```

COM
COMB

Complement dst n051DD



Operation: (dst)←~(dst)

Condition Codes:
N: set if most significant bit of result is set; cleared otherwise
Z: set if result is 0; cleared otherwise
V: cleared
C: set

Description: Replaces the contents of the destination address by their logical complement (each bit equal to 0 is set and each bit equal to 1 is cleared)
Byte: Same

Example:

COM R0

Before	After
(R0) = 013333	(R0) = 164444
N Z V C	N Z V C
0 1 1 0	1 0 0 1

ISP:

```
COM:
   r ← ¬ D'; next                          result is complement of D
      N ← r<15>;                           negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);   zero?
      V ← 0;                               clear V
      C ← 1;                               set C
      D ← r                                transmit result to D

COMB:
   r ← ¬ Db'; next                         result is complement of D
      N ← r<7>;                            negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);    zero?
      V ← 0;                               clear V
      C ← 1;                               set C
      Db ← r                               transmit result to D
```

4.4.2 Shifts

Scaling data by factors of two is accomplished by the shift instructions:

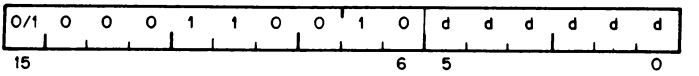
ASR - Arithmetic shift right	ASH - Multiple shift one word
ASL - Arithmetic shift left	ASHC - Multiple shift two words

The sign bit (bit 15) of the operand is replicated in shifts to the right. The low order bit is filled with 0 in shifts to the left. Bits shifted out of the C bit, as shown in the following examples, are lost.

ASR
ASRB

Arithmetic Shift Right dst

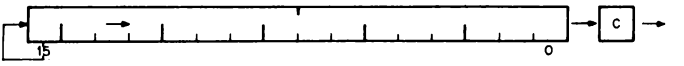
n062DD



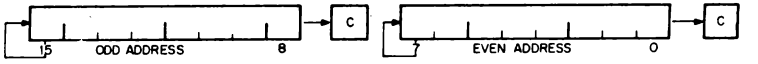
Operation: (dst)←(dst) shifted one place to the right

Condition Codes:
N: set if the high-order bit of the result is set (result < 0); cleared otherwise
Z: set if the result = 0; cleared otherwise
V: loaded from the Exclusive OR of the N-bit and C-bit (as set by the completion of the shift operation)
C: loaded from low-order bit of the destination

Description: Word: Shifts all bits of the destination right one place. Bit 15 is replicated. The C-bit is loaded from bit 0 of the destination. ASR performs signed division of the destination by two.
Word:



Byte:



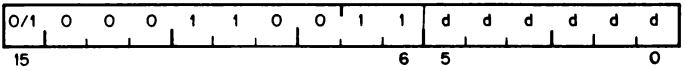
ISP:

```
ASR:
   r ← D'/2; next                          result is D/2
     C ← D<0>;                             carry receives least significant bit
     N ← r<15>;                            negative?
    (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0); next   zero?
        (N ⊕ C) ⇒ (V ← 1 else V ← 0);      overflow is "Exclusive OR" of N and C
         D ← r

ASRB:
   r ← Db'/2; next                         result is D/2
     C ← Db<0>;                            carry receives least significant bit
     N ← r<7>;                             negative?
    (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0); next   zero?
        (N ⊕ C) ⇒ (V ← 1 else V ← 0);      overflow is "Exclusive OR" of N and C
         Db ← r
```

ASL
ASLB

Arithmetic Shift Left dst

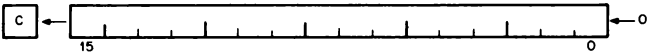
n063DD



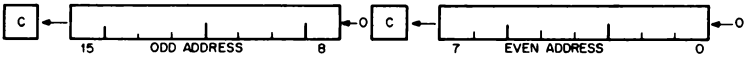
Operation: (dst)←(dst) shifted one place to the left

Condition Codes:
N: set if high-order bit of the result is set (result < 0); cleared otherwise
Z: set if the result = 0; cleared otherwise
V: loaded with the exclusive OR of the N-bit and C-bit (as set by the completion of the shift operation)
C: loaded with the high-order bit of the destination

Description:
Word: Shifts all bits of the destination left one place. Bit 0 is loaded with an 0. The C-bit of the status word is loaded from the most significant bit of the destination. ASL performs a signed multiplication of the destination by 2 with overflow indication.
Byte: Same
Word:



Byte:



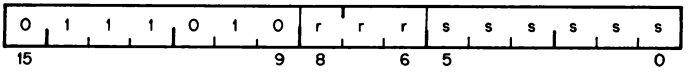
ISP:

```
ASL:
   r ← D'<15>□D'<13:0>□0; next                    result is D*2
      C ← D <14>; next                            bit squeezed out to C
         N ← r<15>;                               negative?
         (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0); next zero?
             (N ⊕ C) ⇒ (V ← 1 else V ← 0);        overflow is "Exclusive OR" of N and C
              D ← r                              transmit result to D
ASLB:
   r ← Db'<7>□Db'<5:0>□0;next                     result is D*2
      C ← Db<6>; next                             bit squeezed out to C
         N ← r<7>;                                negative?
         (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0); next  zero?
             (N ⊕ C) ⇒ (V ← 1 else V ← 0);        overflow is "Exclusive OR" of N and C
              Db ← r                             transmit result to D
```

750 ns + 150 ns x absolute value shift count

ASH

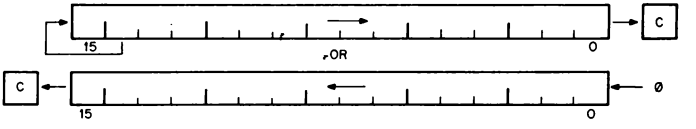
Shift Arithmetically 072RSS



Operation: R⇇ R Shifted arithmetically NN places to right or left
Where NN = low order 6 bits of source

Condition Codes: N: set if result <0; cleared otherwise
Z: set if result =0; cleared otherwise
V: set if sign of register changed during shift; cleared otherwise
C: loaded from last bit shifted out of register

Description: The contents of the register are shifted right or left the number of times specified by the shift count. The shift count is taken as the low order 6 bits of the source operand. This number ranges from -32 to +31. Negative is a a right shift and positive is a left shift.

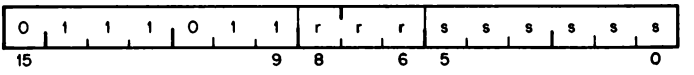


ISP:

```
ASH:
   r<79:0> ← R[sr]<15:0> x 2 ↑ (df<5:0> + 32) mod 64; next      form result
     R[sr]<15:0> ← r<47:32>; next                               store result
    (R[sr] = 0) ⇒ (Z ← 1 else Z ← 0);                           zero?
    (R[sr]<15> = 0) ∧ (r<79:48> = 0) ∨ (R[sr]<15> = 1) ∧        overflow?
        (r<79:48> ≠ -1) ⇒
           (V ← 1 else V ← 0);
     N ← R[sr]<15>;                                             negative?
    (df<5> = 1) ⇒ C ← r<31>;                                    carry?
    (df<5> = 0) ∧ df<5:0> ≠ 0) ⇒ C ← r<48>;
    (df<5:0> = 0) ⇒ C ← 0
```

ASHC

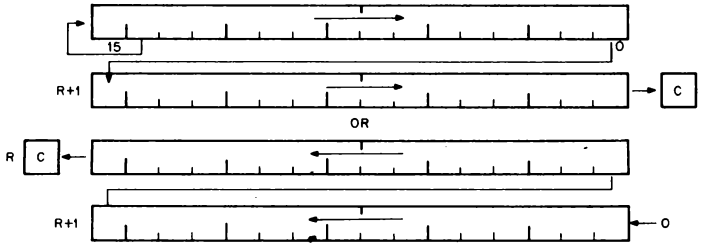
Arithmetic Shift Combined 073RSS



Operation: R, Rv1←R, Rv1 The double word is shifted NN places to the right or left, where NN = low order six bits or source

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result =0; cleared otherwise
V: set if sign bit changes during the shift; cleared otherwise
C: loaded with high order bit when SC>0; loaded with low order bit when SC<0 (loaded with the last bit shifted out of the 32-bit operand)

Description: The contents of the register and the register ORed with one are treated as one 32 bit word, R + 1 (bits 0-15) and R (bits 16-31) are shifted right or left the number of times specified by the shift count. The shift count is taken as the low order 6 bits of the source operand. This number ranges from −32 to +31. Negative is a right shift and positive is a left shift. When the register chosen is an odd number the register and the register OR'ed with one are the same. In this case the right shift becomes a rotate. The 16 bit word is rotated right the number of bits specified by the shift count.



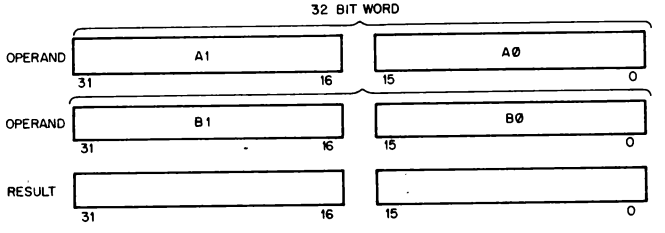
ISP:

```
ASHC:
    r<95:0> ← (R[sr]@R[sr ∨ 1] x 2 ↑ (df<5:0> ? 32) mod 64; next
        R[sr] ← r<63:48>; next
            R[sr] ∨ 1] ← r<47:32>; next
            (R[sr]@R[sr ∨ 1] = 0) ⇒ (Z ← 1 else Z ← 0);          zero?
            N ← R[sr]<15>;                                       negative?
            (r<63> = 0) ∧ (r<95:64> ≠ 0) ∨ (r<63> ≠ 0) ∧ (r<95:64 ≠ -1) ⇒   overflow?
                (V ← 1 else V ← 0);
            (df<5> = 1) ⇒ C ← r<31>;                             carry
            (df<5> = 0) ∧ (df<5:0> ≠ 0) ⇒ C ← r<64>;
            (df<5:0> = 0) ⇒ C ← 0
```

4.4..3 Multiple Precision

It is sometimes necessary to do arithmetic on operands considered as multiple words or bytes. The PDP-11 makes special provision for such operations with the instructions ADC (Add Carry) and SBC (Subtract Carry) and their byte equivalents.

For example two 16-bit words may be combined into a 32-bit double precision word and added or subtracted as shown below:



Example:

The addition of -1 and -1 could be performed as follows:

-1 = 37777777777

(R1) = 177777 (R2) = 177777 (R3) = 177777 (R4) = 177777

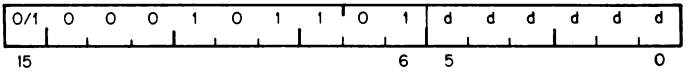
```
ADD   R1,R2
ADC   R3
ADD   R4,R3
```

1. After (R1) and (R2) are added, 1 is loaded into the C bit
2. ADC instruction adds C bit to (R3); (R3) = 0
3. (R3) and (R4) are added
4. Result is 37777777776 or -2

ADC
ADCB

Add Carry dst

n055DD



Operation: (dst)←(dst) + (C)

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result =0; cleared otherwise
V: set if (dst) was 077777 and (C) was 1; cleared otherwise
C: set if (dst) was 177777 and (C) was 1; cleared otherwise

Description: Adds the contents of the C-bit into the destination. This permits the carry from the addition of the low-order words to be carried into the high-order result.
Byte: Same

Example: Double precision addition may be done with the following instruction sequence:

```
ADD   A0,B0      ; add low-order parts
ADC   B1         ; add carry into high-order
ADD   A1,B1      ; add high order parts
```

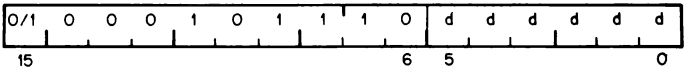
ISP:

```
ADC:
   r ← D' + C; next
      N ← r<15> ;                                   negative?
      (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);           zero?
      (r<15:0> = 100000₈) ∧ (C=1)   (V ← 1 else V ← 0); overflow if largest negative number
      (r<15:0> = 0) ∧ (C=1) ⇒ (C ← 1 else C ← 0);
       D ← r                                        transmit result to D

ADCB:
   r ← Db' + C; next
      N ← r<7>;                                     negative?
      (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);            zero?
      (r<7:0> = 200₈) ∧ (C=1) ⇒ (V ← 1 else V ← 0); overflow if largest negative number
      (r<7:0> = 0) ∧ (C=1) ⇒ (C ← 1 else C ← 0);
       Db ← r                                       transmit result to D
```

300 ns

SBC
SBCB

Subtract Carry dst — n056DD



Operation: (dst)←(dst)−(C)

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result 0; cleared otherwise
V: set if result is 100000; cleared otherwise
C: cleared if result is 0 and C = 1; set otherwise

Description: Word: Subtracts the contents of the C-bit from the destination. This permits the carry from the subtraction of two low-order words to be subtracted from the high order part of the result.
Byte: Same

Example: Double precision subtraction is done by:

```
SUB   A0,B0
SBC   B1
SUB   A1,B1
```

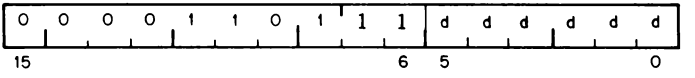
ISP:

```
SBC:
   r ← D' - C; next                                  result is difference of D and C
     N ← r<15>;                                      negative?
    (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);             zero?
    (r<15:0> = 100000₈) ⇒ (V ← 1 else V ← 0);       overflow?
    (r<15:0> = 0) ∧ (C=1) ⇒ (C ← 0 else C ← 1);
     D ← r                                           transmit result to D

SBCB:
   r ← Db' - C; next                                 result is difference of D and C
     N ← r<7>;                                       negative?
    (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);               zero?
    (r<7:0> = 200₈) ⇒ (V ← 1 else V ← 0);            overflow?
    (r<7:0> = 0) ∧ (C=1) ⇒ (C ← 0 else C ← 1);
     Db ← r                                          transmit result to D
```

SXT

Sign Extend dst

0067DD



Operation: (dst)← 0 if N bit is clear
(dst)← -1 N bit is set

Condition Codes: N: unaffected
Z: set if N bit clear
V: unaffected
C: unaffected

Description: If the condition code bit N is set then a -1 is placed in the destination operand: if N bit is clear, then a 0 is placed in the destination operand. This instruction is particularly useful in multiple precision arithmetic because it permits the sign to be extended through multiple words.

Example:

SXT **A**

Before	After
(**A**) = 012345	(**A**) = 177777
N Z V C	N Z V C
1 0 0 0	1 0 0 0

ISP:

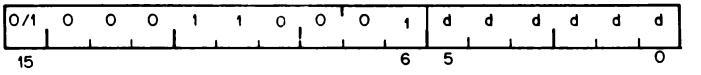
```
SXT:
  (N=1) ⇒ (r<15:0> ← -1 else r<15:0> ← 0); next
   D' ← r;
  (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0)                    zero?
```

4.4.4 Rotates

The rotate instructions operate on the destination word and the C bit as though they formed a 17-bit "circular buffer'. These instructions facilitate sequential bit testing and detailed bit manipulation.

ROL
ROLB

Rotate Left dst n061DD

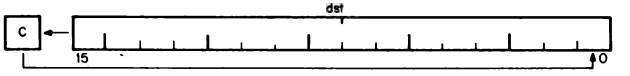


Condition Codes: N: set if the high-order bit of the result word is set (result < 0): cleared otherwise
Z: set if all bits of the result word = 0; cleared otherwise
V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the rotate operation)
C: loaded with the high-order bit of the destination

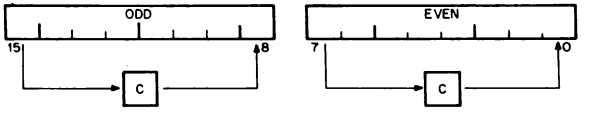
Description: Word: Rotate all bits of the destination left one place. Bit 15 is loaded into the C-bit of the status word and the previous contents of the C-bit are loaded into Bit 0 of the destination.
Byte: Same

Example:

Word:



Bytes:



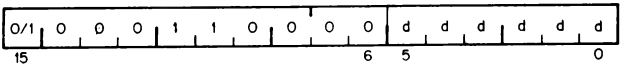

```
ROL:
   r<16:0> ← D'<15:0>□C; next                    result is D and C rotated
      N ← r<15>;                                 negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);         zero?
      C□D ← r; next                             transmit result to C and D
         (N ⊕ C) ⇒ (V ← 1 else V ← 0)            V is based on new result of N and C
ROLB:
   r<8:0> ← Db'<7:0>□C; next                    result is D and C rotated
      N ← r<7>;                                 negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);         zero?
      C□Db ← r; next                            transmit result to C and D
         (N ⊕ C) ⇒ (V ← 1 else V ← 0)            V is based on new result of N and C
```

300 ns

ROR
RORB

Rotate Right dst

n060DD

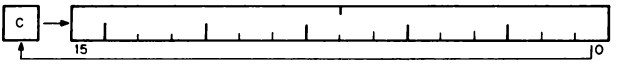


Condition Codes: N: set if the high-order bit of the result is set (result < 0); cleared otherwise
Z: set if all bits of result = 0; cleared otherwise
V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the rotate operation)
C: loaded with the low-order bit of the destination

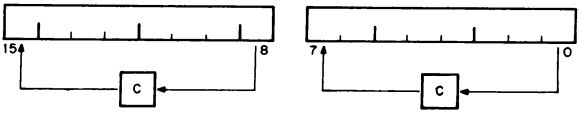
Description: Rotates all bits of the destination right one place. Bit 0 is loaded into the C-bit and the previous contents of the C-bit are loaded into bit 15 of the destination.
Byte: Same

Example:

Word:



Byte:

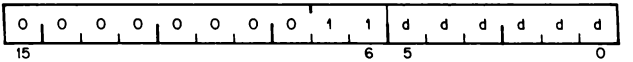



```
ROR:
   r<16:0> ← D<0>□C□D'<15:1>; next                  result is D and C rotated
      N ← r<15>;                                    negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);           zero?
      C□D<15:0> ← r; next                          transmit result to C and D
         (N ⊕ C) ⇒ (V ← 1 else V ← 0)               V is based on new result of N and C

RORB:
   r<8:0> ← Db'<0>□C□Db'<7:1>; next                 result is D and C rotated
      N ← r<7>;                                     negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);            zero?
      C□Db ← r; next                               transmit result to C and D
         (N ⊕ C) ⇒ (V ← 1 else V ← 0)               V is based on new result of N and C
```

SWAB

Swap Bytes dst

0003DD



Operation: Byte 1/Byte 0 ← Byte 0/Byte 1

Condition Codes:
N: set if high-order bit of low-order byte (bit 7) of result is set; cleared otherwise
Z: set if low-order byte of result = 0; cleared otherwise
V: cleared
C: cleared

Description: Exchanges high-order byte and low-order byte of the destination word (destination must be a word address).

Example:

SWAB R1

Before	After
(R1) = 077777	(R1) = 177577
N Z V C	N Z V C
1 1 1 1	0 0 0 0

ISP:

```
r ← D'<7:0>@D'<15:8>; next                    result is byte swapped of D
   N ← r<7>;                                  negative?
  (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);          zero?
   V ← 0;                                     clear V, C
   C ← 0;
       D ← r                                  transmit result to D
```

4.5 DOUBLE OPERAND INSTRUCTIONS

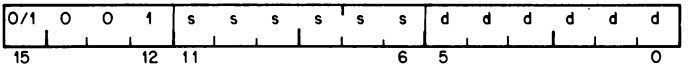
Double operand instructions provide an instruction (and time) saving facility since they eliminate the need for "load"and "save" sequences such as those used in accumulator-oriented machines.

General:	MOV MOVB	ADD	SUB	CMP CMPB
Register Destination:	MUL	DIV	XOR	
Logical:	BIS BISB	BIT BITB	BIC BICB	

4.5.1 Double Operand General Instructions

MOV
MOVB

Mov src. dst n1SSDD



Operation: (dst)←(src)

Condition Codes:
N: set if (src) <0; cleared otherwise
Z: set if (src) = 0; cleared otherwise
V: cleared
C: not affected

Description: Word: Moves the source operand to the destination location. The previous contents of the destination are lost. The contents of the source address are not affected.
Byte: Same as MOV. The MOVB to a register (unique among byte instructions) extends the most significant bit of the low order byte (sign extension). Otherwise MOVB operates on bytes exactly as MOV operates on words.

Example:

```
MOV    XXX,R1          ; loads Register 1 with the contents of memory location; XXX represents a programmer-defined mnemonic used to represent a memory location

MOV    #20,R0          ; loads the number 20 into Register 0; "#"indicates that the value 20 is the operand

MOV @#20,-(R6)         ; pushes the operand contained in location 20 onto the stack

MOV    (R6)+,@ #177566 ; pops the operand off a stack and moves it into memory location 177566 (terminal print buffer)
```

MOV R1,R3 ; performs an interregister transfer

MOVB @#177562,@#177566 ; moves a character from terminal keyboard buffer to terminal buffer

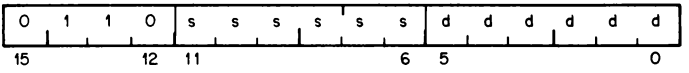
ISP:

```
MOVE:
    r.l ← S.a; next                          ! move source to intermediate result register, r
        N ← r<15>;                           ! negative?
        (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);  ! zero - if 16 bits of r are all zero then Z is set
                                             !  to 1 else Z is set to 0
        V ← 0;                               ! overflow is cleared
        D.l ← r                              ! transmit result to destination

MOVB:
    r ← Sb'; next                            ! move source to intermediate result
        N ← r<7>;                            ! negative?
        (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);   ! zero?
        V ← 0;                               ! clear V
        Db' ← r                              ! transmit result to Db
```

ADD

Add src, dst

06SSDD



Operation: (dst)←(src) + (dst)

Condition Codes: N: set if result <0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: set if there was arithmetic overflow as a result of the operation; that is both operands were of the same sign and the result was of the opposite sign; cleared otherwise
C: set if there was a carry from the most significant bit of the result; cleared otherwise

Description: Adds the source operand to the destination operand and stores the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. Two's complement addition is performed.

Examples:

Add to register:	ADD	20,R0
Add to memory:	ADD	R1,XXX
Add register to register:	ADD	R1,R2
Add memory to memory:	ADD@	# 17750,XXX

XXX is a programmer-defined mnemonic for a memory location.

ISP:

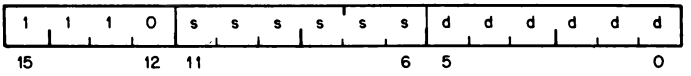
```
ADD:
   r<16:0> ← S' + D'; next                    determine intermediate result sum of 17 bits
      N ← r<15>;                              negative?
      (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);     zero?
      (S<15> ≡ D<15>) ∧ (S<15> ⊕ r<15>) ⇒ (    overflow? if signs of operands agree and sign of
          V ← 1 else V ← 0);                   an operand and the sign of the result disagree
                                               then set V to 1 else set V to 0
      C ← r<16>;                              carry the 17th bit
      D ← r                                   transmit result to D
```

300 ns

SUB

Subtract src. dst

16SSDD



Operation: (dst)←(dst)−(src) [in detail (dst)←(dst) + ~(src) + 1]

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result =0; cleared otherwise
V: set if there was arithmetic overflow as a result of the operation, that is if operands were of opposite signs and the sign of the source was the same as the sign of the result; cleared otherwise
C: cleared if there was a carry from the most significant bit of the result; set otherwise

Description: Subtracts the source operand from the destination operand and leaves the result at the destination address. The orignial contents of the destination are lost. The contents of the source are not affected. In double-precision arithmetic the C-bit, when set, indicates a "borrow".

Example:

SUB R1,R2

Before	After
(R1) = 011111	(R1) = 011111
(R2) = 012345	(R2) = 001234
N Z V C	N Z V C
1 1 1 1	0 0 0 0

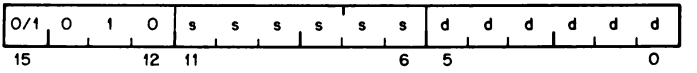
ISP:

```
SUB:
   r ← D' - S'; next                          17 bit result is D minus S; actually r ← ¬ S+D+1;
      N ← r<15>;                              negative?
      (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);     zero?
      (D<15> ≡ ¬ S<15>) ∧ (D<15> ∈ r<15>) ⇒ (  overflow? (see add)
         V ← 1 else V ← 0);
      C ← r<16>;                              borrow from 17th bit
      D ← r                                   move result to D
```

CMP
CMPB

Compare src, dst

n2SSDD



Operation: (src)-(dst) [in detail, (src) + ~ (dst) + 1]

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result =0; cleared otherwise
V: set if there was arithmetic overflow; that is, operands were of opposite signs and the sign of the destination was the same as the sign of the result; cleared otherwise
C: cleared if there was a carry from the most significant bit of the result; set otherwise

Description: Compares the source and destination operands and sets the condition codes, which may then be used for arithmetic and logical conditional branches. Both operands are unaffected. The only action is to set the condition codes. The compare is customarily followed by a conditional branch instruction. Note that unlike the subtract instruction the order of operation is (src)-(dst), not (dst)-(src).

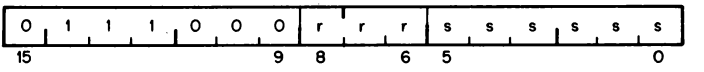
ISP:

```
CMP:
   r ← S' - D'; next                          compare affects CC only
     N ← r<15>;                               negative?
    (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);       zero?
    (S<15> ≡ ¬ D<15>) ∧ (S<15> ⊕ r<15>) ⇒ (   overflow? (see add)
        V ← 1 else V ← 0);
        C ← r<16>                             17th bit is carry


CMPB:
   r<8:0> ← Sb' - Db'; next                   compare affects CC only
     N ← r<7>;                                negative?
    (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);        zero?
    (Sb<7> ≡ ¬ Db<7>) ∧ (Sb<7> ∈ r<7>) ⇒ (    overflow? (see add)
        V ← 1 else V ← 0);
     C ← r<8>                                 8th bit is carry
```

3.3 μs

MUL

Multiply 070RSS



Operation: R, Rv1 ← R x(src)

Condition Codes:
N: set if product is <0; cleared otherwise
Z: set if product is 0; cleared otherwise
V: cleared
C: set if the result is less than -2^{15} or greater than or equal to $2^{15}-1$.

Description: The contents of the destination register and source taken as two's complement integers are multiplied and stored in the destination register and the succeeding register (if R is even). If R is odd only the low order product is stored. Assembler syntax is : MUL S,R.
(Note that the actual destination is R, Rv1 which reduces to just R when R is odd.)

Example: 16-bit product (R is odd)

```
CLC              ;Clear carry condition code
MOV  #400,R1
MUL  #10,R1
BCS ERROR          ;Carry will be set if
                   ;product is less than
                   ;-2^15 or greater than or equal to 2^15
                   ;no significance lost
```

Before	After
(R1) = 000400	(R1) = 004000

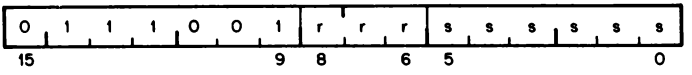
ISP:

```
r<31:0> ← D' x R[sr]; next                          ! form the 32-bit product of the destination and a register
R[sr]<15:0> ← r<31:16>; next                        ! store the high order result in the register
R[sr ∨ 1]<15:0> ← r<15:0>;                          ! store the low order result in an even numbered register
(r<31:0> = 0) ⇒ (Z ← 1 else Z ← 0);                 ! zero?
N ← r<31>;                                          ! negative?
(r<31:0> < -2^15) ∨ (r<31:0> ≥ 2^15) ⇒              ! set carry if more than 16 significant bits in result
    (C ← 1 else C ← C);
V ← 0                                               ! no overflow possible
```

DIV

Divide

071RSS



Operation: R, Rv1 ← R, Rv1 /(src)

Condition Codes:
N: set if quotient <0; cleared otherwise
Z: set if quotient =0; cleared otherwise
V: set if source =0 or if the absolute value of the register is larger than the absolute value of the source. (In this case the instruction is aborted because the quotient would exceed 15 bits.)
C: set if divide 0 attempted; cleared otherwise

Description: The 32-bit two's complement integer in R and Rv1 is divided by the source operand. The quotient is left in R; the remainder in Rv1. Division will be performed so that the remainder is of the same sign as the dividend. R must be even.

Example:

```
CLR R0
MOV #20001,R1
DIV #2,R0
```

Before	After	
(R0) = 000000	(R0) = 010000	Quotient
(R1) = 020001	(R1) = 000001	Remainder

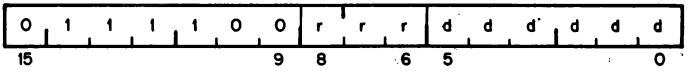
ISP:

```
DIV:
   r1<31:0> ← R[sr]□R[sr ∨ 1]/D'; next                          ! form quotient
      r2<15:0> ← R[sr]□R[sr ∨ 1] - r1×D; next                   ! form remainder
         R[sr] ← r1<15:0>; R[sr ∨ 1] ← r2;                      ! store results
         (r1<31:0> = 0) ⇒ (Z ← 1 else Z ← 0);                   ! zero?
         N ← r1<15>;                                            ! negative?
         (D=0) ⇒ (C ← 1 else C ← 0);                            ! carry?
         ((r1<15> = 0) ∧ (r1<31:16> ≠ 0)) ∨ ((r1<15> = 1) ∧     ! overflow possible
            (r1<31:16> ≠ -1)) ∨ (D=0) ⇒
               (V ← 1 else V ← 0)
```

300 ns

XOR

Exclusive Or 074RDD



Operation: (dst)←Rv(dst)

Condition Codes:
N: set if the result <0; cleared otherwise
Z: set if result =0; cleared otherwise
V: cleared
C: unaffected

Description: The exclusive OR of the register and destination operand is stored in the destination address. Contents of register are unaffected. Assembler format is: XOR R,D

Example:

XOR R0,R2

Before	After
(R0) = 001234	(R0) = 001234
(R2) = 001111	(R2) = 000325

ISP:

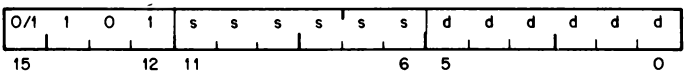
```
XOR:
   r ← R[sr] ∈ D'; next
      R[sr] ← r;
      (r=0: ⇒ Z ← 1 else Z ← 0);          zero?
      N ← r<15>;                          negative?
      V ← 0                               clear V
```

4.5.2 Logical Instructions

These instructions have the same format as the double operand arithmetic group. They permit operations on data at the bit level.

BIS
BISB

Bit Set src dst



Operation: (dst)←(src) v (dst)

Condition Codes:
N: set if high-order bit or result set, cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

Description: Performs "Inclusive OR" operation between the source and destination operands and leaves the result at the destination address; that is, corresponding bits set in the source are set in the destination. The contents of the destination are lost.

Example:

BIS R0,R1

Before	After
(R0) = 001234	(R0) = 001234
(R1) = 001111	(R1) = 001335
N Z V C	N Z V C
0 0 0 0	0 0 0 0

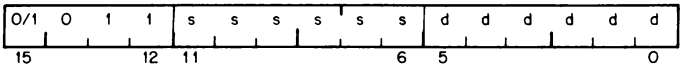
ISP:

```
BIS:
   r ← D' v S'; next                          result is S "OR" D
      N ←r<15>;                               negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);      zero?
      V ← 0;                                  clear V
      D ← r                                   transmit result to D


BISB:
   r ← Db' v Sb'; next                        result is S "OR" D
      N ← r<7>;                               negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);       zero?
      V ← 0;                                  clear V
      Db ← r                                  transmit result to D
```

BIT
BITB

Bit Test src, dst

n3SSDD



Operation: (dst) Λ (src)

Condition Codes:
N: set if high-order bit of result set; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

Description: Performs logical "and" comparison of the source and destination operands and modifies condition codes accordingly. Neither the source nor destination operands are affected. The BIT instruction may be used to test whether any of the corresponding bits that are set in the destination are also set in the source or whether all corresponding bits set in the destination are clear in the source.

Example:

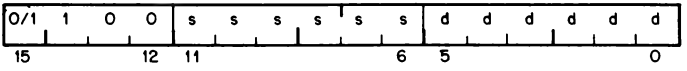
```
BIT    #30,R3      ; test bits 3 and 4 of R3 to see
                   ; if both are off
BEQ    HELP        ; BEQ to HELP will occur if
                     both are off
```

ISP:

```
BIT:
   r ← D' ∧ S'; next                          test result is "AND" of D and S
      N ← r<15>;                              negative?
     (r<15: 0> = 0) ⇒ (Z ← 1 else Z ← 0);     zero?
      V ← 0                                   clear V

BITB:
   r ← Db' ∧ Sb'; next                        test result is "AND" of D and S
      N ← r<7>;                               negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);       zero?
      V ← 0                                   clear V
```

BIC
BICB

Bit Clear src dst n4SSDD



Operation: (dst)←~(src)Λ(dst)

Condition Codes:
N: set if high order bit of result set; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

Description: Clears each bit in the destination that corresponds to a set bit in the source. The original contents of the destination are lost. The contents of the source are unaffected.

Example:

BIC R3,R4

Before	After
(R3) = 001234	(R3) = 001234
(R4) = 001111	(R4) = 000101
N Z V C	N Z V C
1 1 1 1	0 0 0 1

ISP:

```
BIC:
   r ← D' ∧ ¬ S'; next                          result is D "AND" "NOT" S
      N ← r<15>;                                negative?
     (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);        zero?
      V ← 0;                                    clear V
      D ← r                                     transmit result to D


BICB:
   r ← Db' ∧ ¬ Sb'; next                        result is D "AND" "NOT" S
      N ← r<7>;                                 negative?
     (r<7:0> = 0) ⇒ (Z ← 1 else Z ← 0);         zero?
      V ← 0;                                    clear V
      Db ← r                                    transmit result to D
```

4.6 PROGRAM CONTROL INSTRUCTIONS

4.6.1 Branches

The instruction causes a branch to a location defined by the sum of the offset (multiplied by 2) and the current contents of the Program Counter if:

a) the branch instruction is unconditional

b) it is conditional and the conditions are met after testing the condition codes (status word).

The offset is the number of words from the current contents of the PC. Note that the current contents of the PC point to the word following the branch instruction.

Although the PC expresses a byte address, the offset is expressed in words. The offset is automatically multiplied by two to express bytes before it is added to the PC. Bit 7 is the sign of the offset. If it is set, the offset is negative and the branch is done in the backward direction. Similarly if it is not set, the offset is positive and the branch is done in the forward direction.

The 8-bit offset allows branching in the backward direction by 200$_{8}$ words (400$_{8}$ bytes) from the current PC, and in the forward direction by 177$_{8}$ words (376$_{8}$ bytes) from the current PC.

The PDP-11 assembler handles address arithmetic for the user and computes and assembles the proper offset field for branch instructions in the form:

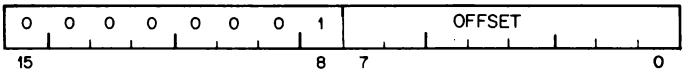
Bxx loc

Where "Bxx" is the branch instruction and "loc" is the address to which the branch is to be made. The assembler gives an error indication in the instruction if the permissable branch range is exceeded. Branch instructions have no effect on condition codes.

600 ns

BR

Branch (unconditional) 0004 loc



Operation: PC ← PC + (2 x offset)

Description: Provides a way of transferring program control within a range of -128 to +127 words with a one word instruction.

Example:

```
        001000    BR xxx
        001002
        001004
xxx:    001006
        001010
```

ISP:

```
BR:
   PC ← PC + sign-extend(instruction<7:0>)⁴          always branch
```

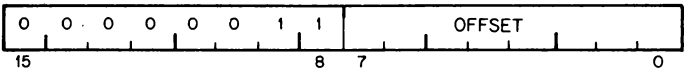
Simple Conditional Branches

BEQ
BNE
BMI
BPL
BCS
BCC
BVS
BVC

300 ns - no branch
600 ns - branch

BEQ

Branch on Equal (zero) 0014 offset



Operation: PC ← PC + (2 x offset) if Z = 1

Condition Codes: Unaffected

Description: Tests the state of the Z-bit and causes a branch if Z is set. As an example, it is used to test equality following a CMP operation, to test that no bits set in the destination were also set in the source following a BIT operation, and generally, to test that the result of the previous operation was zero.

Example:

```
CMP   A,B          ; compare A and B
BEQ   C            ; branch if they are equal
```

will branch to C if A = B (A - B = 0)
and the sequence

```
ADD   A,B          ; add A to B
BEQ   C            ; branch if the result = 0
```

will branch to C if A + B = 0.

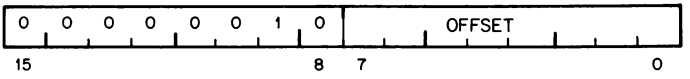
ISP:

```
BEQ:
   (Z=1) =>6 (PC ← PC + sign-extend (instruction<7:0>))      if Z is 1 then branch
```

300 ns - no branch
600 ns - branch

BNE

Branch Not Equal (Zero) 0010 offset



Operation: PC ← PC + (2 x offset) if Z = 0

Condition Codes: Unaffected

Description: Tests the state of the Z-bit and causes a branch if the Z-bit is clear. BNE is the complementary operation to BEQ. It is used to test inequality following a CMP, to test that some bits set in the destination were also in the source, following a BIT, and generally, to test that the result of the previous operation was not zero.

Example:

```
CMP   A,B          ; compare A and B
BNE                ; branch if they are not equal
```

will branch to C if A ≠ B

and the sequence

```
ADD   A,B          ; add A to B
BNE   C            ; Branch if the result not equal
                     to 0
```

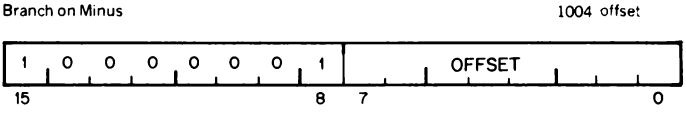
will branch to C if A + B ≠ 0

ISP:

```
BNE:
    (Z=0) ⇒ (PC ← PC + sign-extend(instruction<7:0>))        if Z is 0 then branch
```

300 ns - no branch
600 ns - branch

BMI



Operation: PC ← PC + (2 x offset) if N = 1

Condition Codes: Unaffected

Description: Tests the state of the N-bit and causes a branch if N is set. It is used to test the sign (most significant bit) of the result of the previous operation).

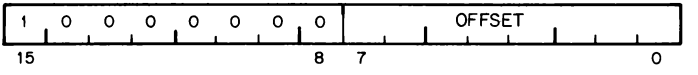
ISP:

```
BMI:
   (N=1) ⇒ (PC ← PC + sign-extend(instruction<7:0>))          if N=1 then branch
```

300 ns - no branch
600 ns - branch

BPL

Branch on Plus 1000 offset



Operation: PC ← PC + (2 x offset) if N = 0

Description: Tests the state of the N-bit and causes a branch if N is clear. BPL is the complementary operation of BMI.

ISP:

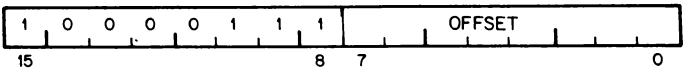
```
BPL:
   (N=0) ⇒ (PC ← PC + sign-extend(instruction<7:0>))          if N=0 then branch
```

300 ns - no branch
600 ns - branch

BCS

Branch on Carry Set

1034 offset



Operation: PC ← PC + (2 x offset) if C = 1

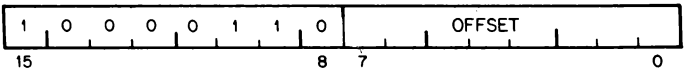
Description: Tests the state of the C-bit and causes a branch if C is set. It is used to test for a carry in the result of a previous operation.

300 ns - no branch
600 ns - branch

BCC

Branch on Carry Clear

1030 offset



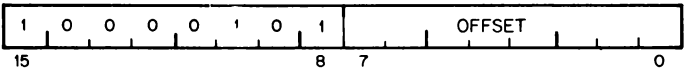
Operation: PC ← PC + (2 x offset) if C = 0

Description: Tests the state of the C-bit and causes a branch if C is clear. BCC is the complementary operation to BCS

300 ns - no branch
600 ns - branch

BVS

Branch on Overflow Set 1024 offset



Operation: PC ← PC + (2 x offset) if V = 1

Description: Tests the state of V bit (overflow) and causes a branch if the V bit is set. BVS is used to detect arithmetic overflow in the previous operation.

ISP:

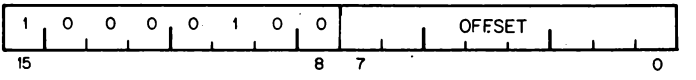
```
BVS:
   (V=1) ⇒ (PC ← PC + sign-extend(instruction<7:0>))          if V=1 then branch
```

300 ns - no branch
600 ns - branch

BVC

Branch on Overflow Clear

1020 offset



Operation: PC ← PC + (2 x offset) if V = 0

Description: Tests the state of the V bit and causes a branch if the V bit is clear. BVC is complementary operation to BVS.

ISP:

```
BVC:
    (V=0) ⇒ (PC ← PC + sign-extend(instruction<7:0>))          if V=0 then branch
```

Signed Conditional Branches
Particular combinations of the condition code bits are tested with the signed conditional branches. These instructions are used to test the results of instructions in which the operands were considered as a signed (two's complement) values.

Note that the sense of signed comparisons differs from that of unsigned comparisons in that in signed 16-bit, two's complement arithmetic the sequence of values is as follows:

largest	077777
	077776
positive	.
	.
	.
	000001
	000000
	177777
	177776
	.
negative	.
	.
	100001
smallest	100000

whereas in unsigned 16-bit arithmetic the sequence is considered to be

highest	177777
	.
	.
	.
	.
	.
	.
	000002
	000001
lowest	000000

The signed conditional branch instructions are:

BLT BGE

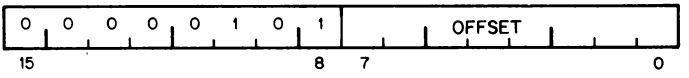
BLE BGT

300 ns - no branch

600 ns - branch

BLT

Branch on Less Than (Zero) 0024 offset



Operation: PC ← PC + (2 x offset) if N ∨ V = 1

Description: Causes a branch if the "Exclusive Or"of the N and V bits are 1. Thus BLT will always branch following an operation that added two negative numbers, even if overflow occurred. In particular, BLT will always cause a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BLT will never cause a branch when it follows a CMP instruction operating on a positive source and negative destination. BLT will not cause a branch if the result of the previous operation was zero (without overflow).

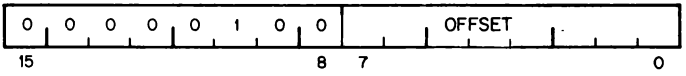
ISP:

```
BLT:
    (N ⊕ V) ⇒ PC ← PC + sign-extend(instruction<7:0>))        if N "Exclusive-OR" V then branch
```

300 ns - no branch
600 ns - branch

BGE

Branch on Greater than or Equal (zero) 0020 offset



Operation: PC ← PC + (2 x offset) if N ∨ V = 0

Description: Causes a branch if N and V are either both clear or both set. BGE is the complementary operation to BLT. Thus BGE will always cause a branch when it follows an operation that caused addition to two positive numbers. BGE will also cause a branch on a zero result.

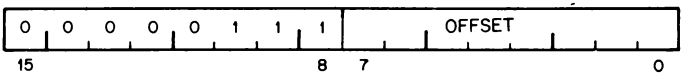
ISP:

```
BGE:
    (N ≡ V) ⇒ (PC ← PC + sign-extend(instruction<7:0>))      ! if N is equivalent to V then branch
```

300 ns - no branch
600 ns - branch

BLE

Branch on Less than or Equal (zero) 0034 offset



Operation: PC ← PC + (2 x offset) if Z v(N ⊻ V) = 1

Description: Operation is similar to BLT but in addition will cause a branch if the result of the previous operation was zero.

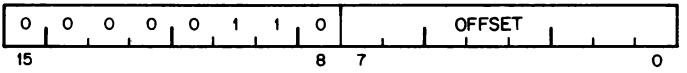
ISP:

```
BLE:
    (Z v (N ⊕ V)) ⇒ (PC ← PC + sign-extend(instruction<7:0>)) if Z "OR" N "Exclusive-OR" V then
                                                              branch
```

300 ns - no branch
600 ns - branch

BGT

Branch on Greater Than (zero) 0030 offset



Operation: PC ← PC + (2 x offset) if Z v(N ⊻ V) = 0

Description: Operation of BGT is similar to BGE, except BGT will not cause a branch on a zero result.

ISP:

```
BGT:
    ¬(Z ∨ (N ⊕ V)) ⇒ (PC ← PC + sign-extend(instruction<7:0>)) ;if "NOT" Z or "N" "Exclusive'OR" V
                                                                 then branch
```

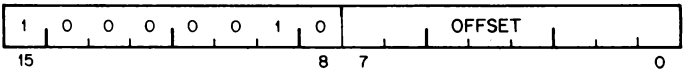
Unsigned Conditional Branches
The Unsigned Conditional Branches provide a means for testing the result of comparison operations in which the operands are considered as unsigned values.

BHI
BLOS
BHIS
BLO

300 ns - no branch
600 ns - branch

BHI

Branch on Higher 1010 offset



Operation: PC ← PC + (2 x offset) if C = 0 and Z = 0

Description: Causes a branch if the previous operation caused neither a carry nor a zero result. This will happen in comparison (CMP) operations as long as the source has a higher unsigned value than the destination.

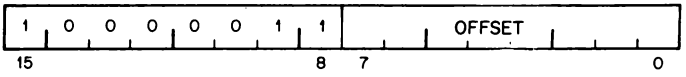
ISP:

```
BHI:
   ¬(C ∨ Z) ⇒ (PC ← PC + sign-extend(instruction<7:0>))      if NOT C "OR" Z then branch
```

300 ns - no branch
600 ns - branch

BLOS

Branch on Lower or Same 1014 offset



Operation: PC ← PC + (2 x offset) if C v Z = 1

Description: Causes a branch if the previous operation caused either a carry or a zero result. BLOS is the complementary operation to BHI. The branch will occur in comparison operations as long as the source is equal to, or has a lower unsigned value than the destination.

ISP:

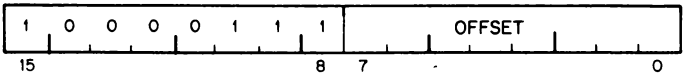
```
BLOS:
   (C ∨ Z) ⇒ (PC ← PC + sign-extend(instruction<7:0>))        if C "OR" Z then branch
```

300 ns - no branch
600 ns - branch

BLO

Branch on Lower

1034 offset



Operation: PC ← PC + (2 x offset) if C = 1

Description: BLO is same instruction as BCS. This mnemonic is included only for convenience.

ISP:

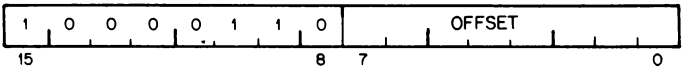
```
BCS/BLO:
    (C=1) ⇒ (PC ← PC + sign-extend(instruction<7:0>))        if C=1 then branch
```

300 ns · no branch
600 ns · branch

BHIS

Branch on Higher or Same

1030 offset



Operation: PC ← PC + (2 x offset) if C = 0

Description: BHIS is the same instruction as BCC. This mnemonic is included only for convenience.

ISP:

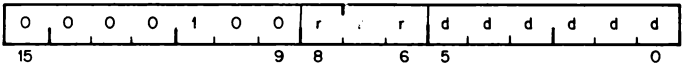
```
BCC/BHIS:
   (C=0) ⇒ (PC ← PC + sign-extend(instruction<7:0>))        if C=0 then branch
```

4.6.2 Subroutine Instructions

The subroutine call in the PDP-11 provides for automatic nesting of subroutines, reentrancy, and multiple entry points. Subroutines may call other subroutines (or indeed themselves) to any level of nesting without making special provision for storage or return addresses at each level of subroutine call. The subroutine calling mechanism does not modify any fixed location in memory, thus providing for reentrancy. This allows one copy of a subroutine to be shared among several interrupting processes. For more detailed description of subroutine programming see Chapter 5.

JSR

Jump to Sub Routine 004 reg. dst



Operation: (tmp)←(dst) (tmp is an internal processor register)

↓(SP)←reg (push reg contents onto processor stack)

reg←PC (PC holds location following JSR; this address

PC←(tmp) now put in reg)

Description: In execution of the JSR, the old contents of the specified register (the "LINKAGE POINTER") are automatically pushed onto the processor stack and new linkage information placed in the register. Thus subroutines nested within subroutines to any depth may all be called with the same linkage register. There is no need either to plan the maximum depth at which any particular subroutine will be called or to include instructions in each routine to save and restore the linkage pointer. Further, since all linkages are saved in a reentrant manner on the processor stack execution of a subroutine may be interrupted, the same subroutine reentered and executed by an interrupt service routine. Execution of the initial subroutine can then be resumed when other requests are satisfied. This process (called nesting) can proceed to any level.

In both JSR and JMP instructions the destination address is used to load the program counter, R7. Thus for example a JSR in destination mode 1 for general register R1 (where (R1) = 100), will access a subroutine at location 100. This is effectively one level less of deferral than operate instructions such as ADD.

A subroutine called with a JSR reg,dst instruction can access the arguments following the call with either autoincrement addressing, (reg) +, (if arguments are accessed sequentially) or by indexed addressing, X(reg), (if accessed in random order). These addressing modes may also be deferred, @(reg) + and @X(reg) if the parameters are operand addresses rather than the operands themselves.

JSR PC, dst is a special case of the PDP-11 subroutine call suitable for subroutine calls that transmit parameters through the general registers. The SP and the PC are the only registers that may be modified by this call.

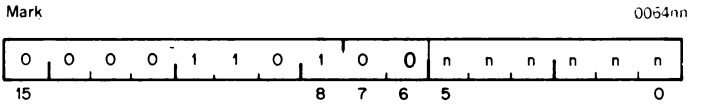
Another special case of the JSR instruction is JSR PC, @(SP)+ which exchanges the top element of the processor stack and the contents of the program counter. Use of this instruction allows two routines to swap program control and resume operation when recalled where they left off. Such routines are called "co-routines."

Return from a subroutine is done by the RTS instruction. RTS reg loads the contents of reg into the PC and pops the top element of the processor stack into the specified register.

ISP:

```
JSR:
   SP ← SP - 2; next
      MW^B[SP] ← R[sr];          stack (push) P[sr];
      R[sr] ← PC;                load P[sr] with Pc
      PC ← Daddress'             jump
```

MARK



Operation: SP←SP + 2xnn nn = number of parameters
PC←R5
R5←(SP)↑

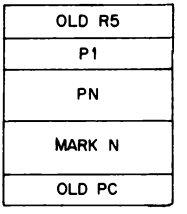
Condition Codes: unaffected

Description: Used as part of the standard PDP-11 subroutine return convention. MARK facilitates the stack clean up procedures involved in subroutine exit. Assembler format is: MARK N

Example:

```
MOV   R5,-(SP)          ;place old R5 on stack
MOV   P1,-(SP)          ;place N parameters
MOV   P2,-(SP)          ;on the stack to be
                        ;used there by the
                        :subroutine
MOV   PN,-(SP)
MOV   #MARKN,-(SP)      ;places the instruction
                        ;MARK N on the stack
MOV   SP ,R5            ;set up address at Mark N in-
                         struction
JSR   PC,SUB            ;jump to subroutine
```

At this point the stack is as follows:



And the program is at the address SUB which is the beginning of the subroutine.

```
SUB:                    ;execution of the subroutine itself

        RTS R5          ;the return begins: this causes
```

the contents of R5 to be placed in the PC which then results in the execution of the instruction MARK N. The contents old PC are placed in R5

MARK N causes: (1) the stack pointer to be adjusted to point to the old R5 value; (2) the value now in R5 (the old PC) to be placed in the PC; and (3) contents of the the old R5 to be popped into R5 thus completing the return from subroutine.

Note: If segmentation is in use a stack must be in I and D spaces (Chapter 6) to execute the MARK instruction

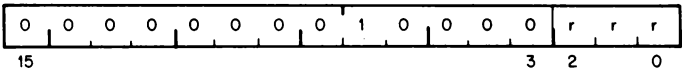
ISP:

```
MARK:
   SP ← PC x df<5:0>; next
      PC ← R[5]; next
         R[5] ← Mw[SP];
         SP ← SP+2
```

1.2 μs

RTS

Return from Subroutine 00020 Reg



Operation: PC←reg
reg← (SP)↑

Description: Loads contents of reg into PC and pops the top element of the processor stack into the specified register.
Return from a non-reentrant subroutine is typically made through the same register that was used in its call. Thus, a subroutine called with a JSR PC, dst exits with a RTS PC and a subroutine called with a JSR R5, dst, may pick up parameters with addressing modes (R5)+, X(R5), or @X(R5) and finally exits, with an RTS R5

ISP:

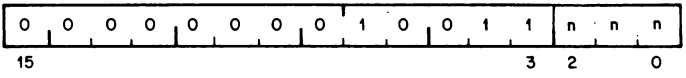
```
RTS:
    PC ← R[dr];                    return ;jump
    R[dr] ← Ms[SP];                unstack (pop) R[dr]
    SP ← SP + 2
```

4.6.3 Program Control Instructions

SPL
JMP
SOB

SPL

Set Priority Level 00023N



Operation: PS (bits 7-5)←Priority

Condition Codes: not affected

Description: The least significant three bits of the instruction are loaded into the Program Status Word (PS) bits 7-5 thus causing a changed priority. The old priority is lost.
Assembler syntax is: SPL N

Note: This instruction is a no op in User and Supervisor modes.

ISP:

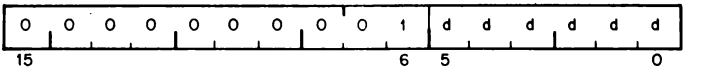
```
SPL:
    PS<7:5> ← df<2:0>
```

600 ns

JMP

Jump

0001DD



Operation: PC←(dst)

Condition Codes: not affected

Description: JMP provides more flexible program branching than provided with the branch instructions. Control may be transferred to any location in memory (no range limitation) and can be accomplished with the full flexibility of the addressing modes, with the exception of register mode 0. Execution of a jump with mode 0 will cause an "illegal instruction"condition. (Program control cannot be transferred to a register.) Register deferred mode is legal and will cause program control to be transferred to the address held in the specified register. Note that instructions are word data and must therefore be fetched from an even-numbered address. A 'boundary error"trap condition will result when the processor attempts to fetch an instruction from an odd address.

Deferred index mode JMP instructions permit transfer of control to the address contained in a selectable element of a table of dispatch vectors.

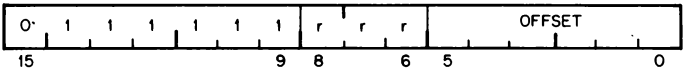
ISP:

```
JMP:
   PC ← Daddress'                    Daddress is computed in a fashion
                                     similar to D
```

750 ns - no branch
600 ns - branch

SOB

Subtract One and Branch 077R offset



Operation: R← R -1 if this result ≠ 0 then PC ← PC -(2 x offset)

Condition Codes: unaffected

Description: The register is decremented. If it is not equal to 0, twice the offset is subtracted from the PC (now pointing to the following word). The offset is interpreted as a sixbit positive number. This instruction provides a fast, efficient method of loop control. Assembler syntax is:

SOB R,A

Where A is the address to which transfer is to be made if the decremented R is not equal to 0. Note that the SOB instruction can not be used to transfer control in the forward direction.

ISP:

```
SOB:
   r ← R[sr] - 1; next                          subtract 1 from EK
      R[sr] ← r;
      (r≠0) ⇒ (PC ← PC-2 x df<5:0>)             branch unequal to zero
```

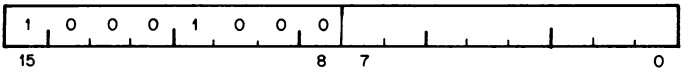
4.6.4 Traps

Trap instructions provide for calls to emulators, I/O monitors, debugging packages, and user-defined interpreters. A trap is effectively an interrupt generated by software. When a trap occurs the contents of the current Program Counter (PC) and Program Status Word (PS) are pushed onto the processor stack and replaced by the contents of a two-word trap vector containing a new PC and new PS. The return sequence from a trap involves executing an RTI or RTT instruction which restores the old PC and old PS by popping them from the stack. Trap vectors are located permanently assigned fixed address.

TRAP
EMT
BPT
IOT
RTI
RTT

EMT

Emulator Traps 104000-104377



Operation:

↓(SP)←PS
↓(SP)←PC
PC←(30)
PS←(32)

Condition Codes:

N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description:

All operation codes from 104000 to 104377 are EMT instructions and may be used to transmit information to the emulating routine (e.g., function to be performed). The trap vector for EMT is at address 30. The new PC is taken from the word at address 30; the new central processor status (PS) is taken from the word at address 32.

Caution: EMT is used frequently by DEC system software and is therefore not recommended for general use.

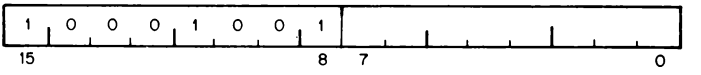
ISP:

```
EMT:
   SP ← SP-2; next                          place
      Mw[SP] ← PS;                           PS and
      SP ← SP-2; next
         Mw[SP] ← PC;                        PC on stack
         PC ← Mw[30];                        take new PC and PS from M[30], M[32]
         PS ← Mw[32]
```

2.25 μs

TRAP

Trap 104400 to 104777



Operation:

↓(SP)←PS
↓(SP)←PC
PC←(34)
PS←(36)

Condition Codes:

N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description:

Operation codes from 104400 to 104777 are TRAP instructions. TRAPs and EMTs are identical in operation, except that the trap vector for TRAP is at address 34.

Note: Since DEC software makes frequent use of EMT, the TRAP instruction is recommended for general use.

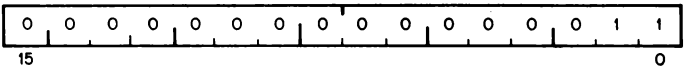
ISP:

```
TRAP:
   SP ← SP-2; next                    place (push)
      Ms[SP] ← PS;                     PS and
      SP ← SP-2; next
         Mw[SP] ← PC;                  PC on stack
         PC ← Mw[34]                   take new PC and PS from M[34], M[36]
         PS ← Mw[36]
```

2.25 μs

BPT

Breakpoint Trap 000003



Operation:

↓(SP)←PS
↓(SP)←PC
PC ← (14)
PS ← (16)

Condition Codes:

N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description:

Performs a trap sequence with a trap vector address of 14. Used to call debugging aids. The user is cautioned against employing code 000003 in programs run under these debugging aids.

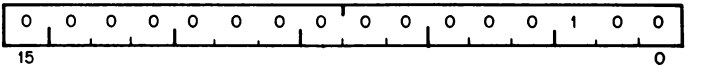
(no information is transmitted in the low byte.)

ISP:

2.25 μs

IOT

I/O Trap 000004



Operation:

↓(SP)←PS
↓(SP)←PC
PC←(20)
PS←(22)

Condition Codes:

N:loaded from trap vector
Z:loaded from trap vector
V:loaded from trap vector
C:loaded from trap vector

Description:

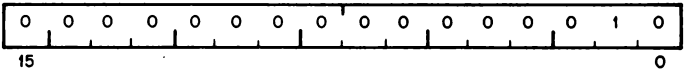
Performs a trap sequence with a trap vector address of 20. Used to call the I/O Executive routine IOX in the paper tape software system, and for error reporting in the Disk Operating System.
(no information is transmitted in the low byte)

ISP:

```
IOT:
   SP ← SP-2; next                    place
      Mw[SP] ← PS;                     PS and
      SP ← SP-2; next
         Mw[SP] ← PC;                  PC on stack
         PC ← Mw[20];                  take new PC and PS from M[20], M[22]
         PS ← Mw[22]
```

RTI

Return from Interrupt 000002



Operation: PC←(SP)↑
PS←(SP)↑

Condition Codes: N: loaded from processor stack
Z: loaded from processor stack
V: loaded from processor stack
C: loaded from processor stack

Description: Used to exit from an interrupt or TRAP service routine. The PC and PS are restored (popped) from the processor stack.

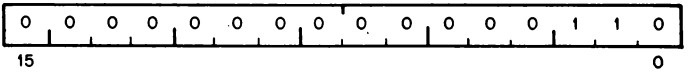
ISP:

```
RTI:
   PC ← Mw[SP];                          unstack (pop) PC for jump
   SP ← SP + 2; next
      PS ← Mw[SP];                       unstack (pop) PS
      SP ← SP + 2
```

1.5 μs

RTT

Return from Interrupt 000006



Operation: PC←(SP)↑
PS←(SP)↑

Condition Codes: N: loaded from processor stack
Z: loaded from processor stack
V: loaded from processor stack
C: loaded from processor stack

Description: This is the same as the RTI instruction except that it inhibits a trace trap, while RTI permits a trace trap. If a trace trap is pending, the first instruction after the RTT will be executed prior to the next "T"trap. In the case of the RTI instruction the "T" trap will occur immediately after the RTI.

Reserved Instruction Traps - These are caused by attempts to execute instruction codes reserved for future processor expansion (reserved instructions) or instructions with illegal addressing modes (illegal instructions). Order codes not corresponding to any of the instructions described are considered to be reserved instructions. JMP and JSR with register mode destinations are illegal instructions. Reserved and illegal instruction traps occur as described under EMT, but trap through vectors at addresses 10 and 4 respectively.

Stack Overflow Trap

Bus Error Traps - Bus Error Traps are:

1. Boundary Errors - attempts to reference instructions or word operands at odd addresses.

2. Time-Out Errors - attempts to reference addresses on the bus that made no response within 5 μs in the PDP-11/45. In general, these are caused by attempts to reference non-existent memory, and attempts to reference non-existent peripheral devices.

Bus error traps cause processor traps through the trap vector address 4.

Trace Trap - Trace Trap enables bit 4 of the PS and causes processor traps at the end of instruction executions. The instruction that is executed after the instruction that set the T-bit will proceed to completion and then cause a processor trap through the trap vector at address 14. Note that the trace trap is a system debugging aid and is transparent to the general programmer.

The following are special cases and are detailed in subsequent paragraphs.

1. The traced instruction cleared the T-bit.

2. The traced instruction set the T-bit.

3. The traced instruction caused an instruction trap.

4. The traced instruction caused a bus error trap.

5. The traced instruction caused a stack overflow trap.

6. The process was interrupted between the time the T-bit was set and the fetching of the instruction that was to be traced.

7. The traced instruction was a WAIT.

8. The traced instruction was a HALT.

9. The traced instruction was a Return from Trap

Note: The traced instruction is the instruction after the one that sets the T-bit.

An instruction that cleared the T-bit - Upon fetching the traced instruction an internal flag, the trace flag, was set. The trap will still occur at the end of execution of this instruction. The stacked status word, however, will have a clear T-bit.

An instruction that set the T-bit - Since the T-bit was already set, setting it again has no effect. The trap will occur.

An instruction that caused an Instruction Trap - The instruction trap is sprung and the entire routine for the service trap is executed. If the service routine exists with an RTI or in any other way restores the stacked status word, the T-bit is set again, the instruction following the traced instruction is executed and, unless it is one of the special cases noted above, a trace trap occurs.

An instruction that caused a Bus Error Trap - This is treated as an Instruction Trap. The only difference is that the error service is not as likely to exit with an RTI, so that the trace trap may not occur.

An instruction that caused a stack overflow - The instruction completes execution as usual - the Stack Overflow does not cause a trap. The Trace Trap Vector is loaded into the PC and PS, and the old PC and PS are pushed onto the stack. Stack Overflow occurs again, and this time the trap is made.

An interrupt between setting of the T-bit and fetch of the traced instruction - The entire interrupt service routine is executed and then the T-bit is set again by the exiting RTI. The traced instruction is executed (if there have been no other interrupts) and, unless it is a special case noted above, causes a trace trap.

Note that interrupts may be acknowledged immediately after the loading of the new PC and PS at the trap vector location. To lock out all interrupts, the PS at the trap vector should raise the processor priority to level 7.

A WAIT - The trap occurs immediately.

A HALT - The processor halts. When the continue key on the console is pressed, the instruction following the HALT is fetched and executed. Unless it is one of the exceptions noted above, the trap occurs immediately following execution.

A Return from Trap - The return from trap instruction either clears or sets the T-bit. It inhibits the trace trap. If the T-bit was set and RTT is the traced instruction the trap is delayed until completion of the next instruction.

Power Failure Trap - is a standard PDP-11 feature. Trap occurs whenever the AC power drops below 95 volts or outside 47 to 63 Hertz. Two milliseconds are then allowed for power down processing. Trap vector for power failure is at locations 24 and 26.

Trap priorities - in case multiple processor trap conditions occur simultaneously the following order of priorities is observed (from high to low):

1. Odd Address
2. Fatal Stack Violation
3. Segment Violation
4. Timeout
5. Parity Error
6. Console Flag
7. Segment Management Trap
8. Warning Stack Violation
9. Power Failure

The details on the trace trap process have been described in the trace trap operational description which includes cases in which an instruction being traced causes a bus error, instruction trap, or a stack overflow trap.

If a bus error is caused by the trap process handling instruction traps, trace traps, stack overflow traps, or a previous bus error, the processor is halted.

If a stack overflow is caused by the trap process in handling bus errors, instruction traps, or trace traps, the process is completed and then the stack overflow trap is sprung.

4.7 MISCELLANEOUS

HALT

WAIT

RESET

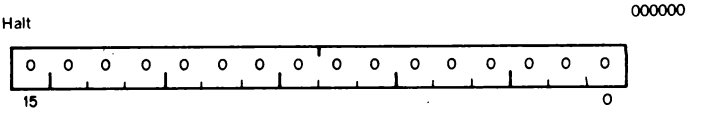
MTPD

MTPI

MFPD

MFPI

HALT



Condition Codes: not affected

Description: Causes the processor operation to cease. The console is given control of the bus. The console data lights display the contents of R0; the console address lights display the address after the halt instruction. Transfers on the UNIBUS are terminated immediately. The PC points to the next instruction to be executed. Pressing the continue key on the console causes processor operation to resume. No INIT signal is given.

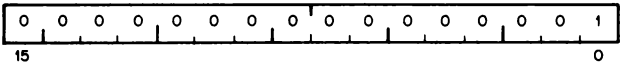
ISP: Note: A halt issued in Supervisor or User Mode will generate a trap.

```
Off ← true                    set activity to Off state
                              no more instructions can
                              be executed until a con-
                              sole action takes place
                              to restart processor
```

WAIT

Wait for Interrupt

000001



Condition Codes: not affected

Description: Provides a way for the processor to relinquish use of the bus while it waits for an external interrupt. Having been given a WAIT command, the processor will not compete for bus use by fetching instructions or operands from memory. This permits higher transfer rates between a device and memory, since no processor-induced latencies will be encountered by bus requests from the device. In WAIT, as in all instructions, the PC points to the next instruction following the WAIT operation. Thus when an interrupt causes the PC and PSW to be pushed onto the processor ation. from the interrupt routine (i.e. execution of an RTI instruction) will cause resumption of the interrupted process at the instruction following the WAIT.

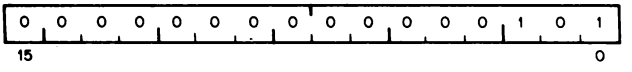
ISP:

```
WAIT:
   Wait ← true                    set activity to Wait state; interrupts
                                  can occur
```

10 ms

RESET

Reset External Bus 000005



Condition Codes: not affected

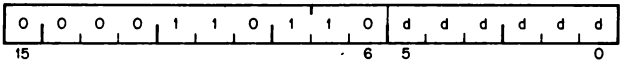
Description: Sends INIT on the UNIBUS for 10 ms. All devices on the UNIBUS are reset to their state at power up.

ISP:

```
Reset:
    Init ← 1;                                   cause a signal, Init, to be one for
                                                 20 milliseconds
    Delay (so milliseconds); next
        Init ← 0
```

MTPI

Move to Previous Instruction Space 0066DD



Operation: (temp)←(SP)↑
(dst)←(temp)

Condition Codes:
N: set if the source <0; otherwise cleared
Z: set if the source =0; otherwise cleared
V: cleared
C: unaffected

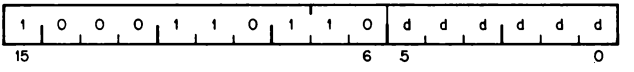
Description: The address of the destination operand is determined in the current address space. MTPI then pops a word off the current stack and stores that word in the destination address in the previous mode's I space (bits 13, 12 of PS).

ISP:

```
r ← Mw[SP];                          get data from stack
SP ← SP+2; next                      pop stack
   D' ← r;                           place at destination
   N ← r<15:>;                       set N, Z and V
  (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);
   V ← 0
```

MTPD

Move to Previous Data Space 1066DD



Operation: (temp)←(SP)↑
(dst)←(temp)

Condition Codes: N: set if the source <0; otherwise cleared
Z: set if the source =0; otherwise cleared
V: cleared
C: unaffected

Description: The address of the destination operand is determined in the current address space as in MTPI. MTPD then pops a word off the current stack and stores that word in the destination address in the previous mode's D space.

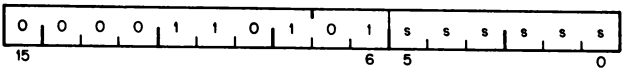
ISP:

```
r ← Mw[SP];                                   get data from stack
SP ← SP+2; next                               pop stack
   D' ← r;                                    place at destination
   N ← r<15>;                                 set N, Z and V
  (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);
   V ← 0
```

1.2 μs

MFPI

Move from Previous Instruction Space 0065SS



Operation: (temp) ← (src)
↑(SP)←(temp)

Condition Codes: N: set if the source <0; otherwise cleared
Z: set if the source =0; otherwise cleared
V: cleared
C: unaffected

Description: This instruction is provided in order to allow inter-address space communication when the PDP11/45 is using the segmentation unit. The address of the source operand is determined in the current address space. That is, the address is determined using the SP and memory segments determined by PS<15:14>The address itself is then used in the previous I space (as determined by PS<13:12>to get the source operand. This operand is then pushed on to the current R6 stack.

ISP:

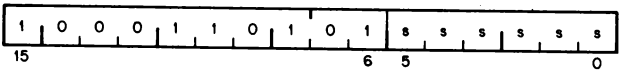
```
r ← S'; next                          get source from previous space
   SP ← SP-2;
   Mw[SP] ← r;                        stack on current address space
   N ← r<15>;                         set N, Z and V
  (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);
   V ← 0
```

1.2 μs

MFPD

Move from Previous Data Space

1065SS



Operation: (temp) ← (src)
(SP) ← (temp)

Condition Codes:
N: set if the source <0; otherwise cleared
Z: set if the source =0; otherwise cleared
V: cleared
C: unaffected

Description: This instruction is provided in order to allow inter-address space communication when the PDP11/45 is using the segmentation unit. The address of the source operand is determined in the current address space. That is, the address is determined using the SP and memory segments determined by PS<15:14>The address itself is then used in the previous D space (as determined by PS<13:12>to get the source operand. This operand is then pushed on to the current R6 stack.

ISP:

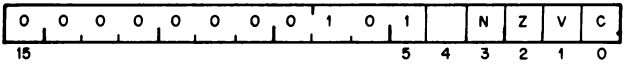
```
r ← S'; next                                    get source from previous space
   SP ← SP-2;
   Mw[SP] ← r;                                  stack on current address space
   N ← r<15>;                                   set N, Z and V
  (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);
   V ← 0
```

4.9 Condition Code Operators

600 ns

CLN SEN
CLZ SEZ
CLV SEV
CLC SEC

Condition Code Operators 0002XX



Description: Set and clear condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (Bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator. i.e. set the bit specified by bit 0, 1, 2 or 3, if bit 4 is a 1. Clear corresponding bits if bit 4 = 0.

Mnemonic	Operation	OP Code
CLC	Clear C	000241
CLV	Clear V	000242
CLZ	Clear Z	000244
CLN	Clear N	000250
SEC	Set C	000261
SEV	Set V	000262
SEZ	Set Z	000264
SEN	Set N	000270
SCC	Set all CC's	000277
CCC	Clear all CC's	000257
	Clear V and C	000243
	No Operation	000240

Combinations of the above set or clear operations may be ORed together to form combined instructions.

CHAPTER 5

ADVANCED PROGRAMMING TECHNIQUES

In order to produce programs which fully utilize the power and flexibility of the PDP-11/45, the reader should become familiar with the various programming techniques which are part of the basic design philosophy of the PDP-11. Although it is possible to program the PDP-11/45 along traditional lines such as "accumulator orientation" this approach does not fully exploit the architecture and instruction set of the PDP-11/45.

5.1 THE STACK

A "stack", as used on the PDP-11, is an area of memory set aside by the programmer for temporary storage or subroutine/interrupt service linkage. The instructions which facilitate "stack" handling are useful features not normally found in low-cost computers. They allow a program to dynamically establish, modify, or delete a stack and items on it. The stack uses the "last-in, first-out" concept, that is, various items may be added to a stack in sequential order and retrieved or deleted from the stack in reverse order. On the PDP-11, a stack starts at the highest location reserved for it and expands linearly downward to the lowest address as items are added to the stack.

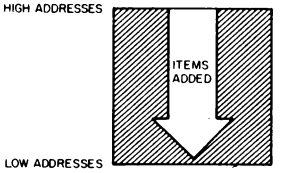


Figure 5-1: Stack Addresses

To keep track of the last item added to the stack (or "where we are" in the stack) a General Register always contains the memory address where the last item is stored in the stack. In the PDP-11 any register except Register 7 (the Program Counter-PC) may be used as a "stack pointer" under program control; however, instructions associated with subroutine linkage and interrupt service automatically use Register 6 (R6) as a hardware "Stack Pointer." For this reason R6 is frequently referred to as the system "SP."

Stacks in the PDP-11 may be maintained in either full word or byte units. This is true for a stack pointed to by any register except R6, which must be organized in full word units only.

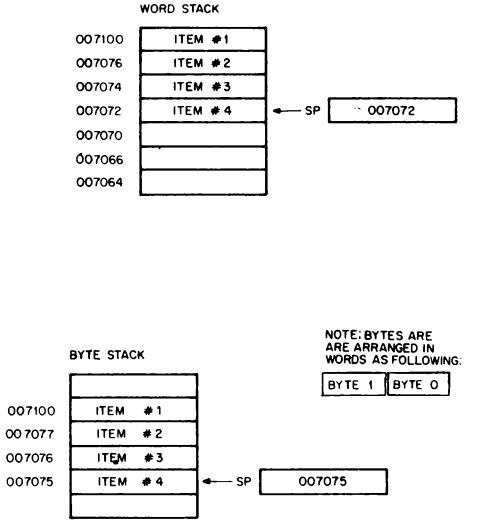


Figure 5-2: Word and Byte Stacks

Items are added to a stack using the autodecrement addressing mode with the appropriate pointer register. (See Chapter 3 for description of the autoincrement/decrement modes).

This operation is accomplished as follows;

```
MOV Source,-(SP)        ;MOV Source Word onto the stack

                        or

MOVB Source,-(SP)       ;MOVB Source Byte onto the stack
```

This is called a "push" because data is "pushed onto the stack."

To remove an item from stack the autoincrement addressing mode with the appropriate SP is employed. This is accomplished in the following manner:

```
MOV (SP)+,Destination      ;MOV Destination Word off the stack

                           or

MOVB (SP)+,Destination     ;MOVB Destination Byte off the stack
```

Removing an item from a stack is called a "pop" for "popping from the stack." After an item has been "popped," its stack location is considered free and available for other use. The stack pointer points to the last-used location implying that the next (lower) location is free. Thus a stack may represent a pool of shareable temporary storage locations.

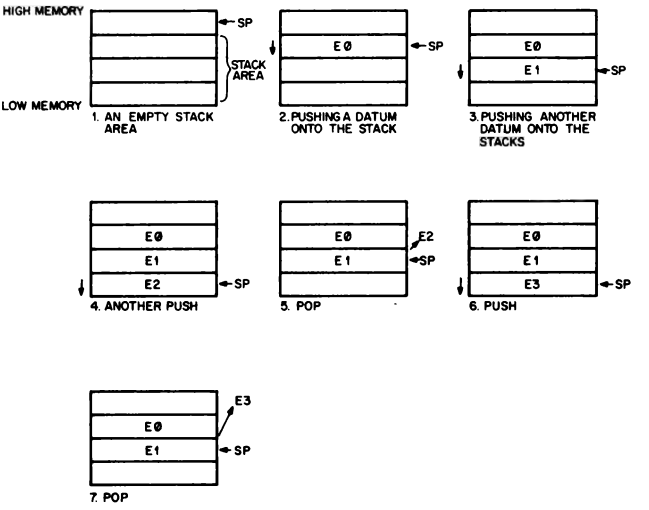


Figure 5-3: Illustration of Push and Pop Operations

As an example of stack usage consider this situation: a subroutine (SUBR) wants to use registers 1 and 2, but these registers must be returned to the calling program with their contents unchanged. The subroutine could be written as follows:

```
Address         Octal Code          Assembler Syntax

076322          010167    SUBR:     MOV R1,TEMP1 ;save R1
076324          000074              *
076326          010267              MOV R2,TEMP2 ;save R2
076330          000072              *
.               .                   .
.               .                   .
.               .                   .
.               .                   .
076410          016701              MOV TEMP1, R1 ;Restore R1
076412          000006              *
076414          016702              MOV TEMP2, R2 ;Restore R2
076416          000002              *
076420          000207              RTS PC
076422          000000              TEMP1: 0
076424          000000              TEMP2: 0
```

*Index Constants

Figure 5-4: Register Saving Without the Stack

OR: Using the Stack

```
Address         Octal Code          Assembler Syntax

010020          010143    SUBR:     MOV R1, -(R3) ;push R1
010022          010243              MOV R2, -(R3) ;push R2
.               .                   .
.               .                   .
.               .                   .
.               .                   .
010130          012301              MOV (R3)+, R2 ;pop R2
010132          012302              MOV (R3)+, R1 ;pop R1
010134          000207              RTS PC
```

Note: In this case R3 was used as a Stack Pointer.

Figure 5-5: Register Saving using the Stack

The second routine uses four less words of instruction code and two words of temporary "stack" storage. Another routine could use the same stack space at some later point. Thus, the ability to share temporary storage in the form of a stack is a very economical way to save on memory usage.

As a further example of stack usage, consider the task of managing an input buffer from a terminal. As characters come in, the terminal user may wish to delete characters from his line; this is accomplished very easily by maintaining a byte stack containing the input characters. Whenever a backspace is received a character is "popped" off the stack and eliminated from consideration. In this example, a programmer has the choice of "popping" characters to be eliminated by using either the MOVB (MOVE BYTE) or INC (INCREMENT) instructions.

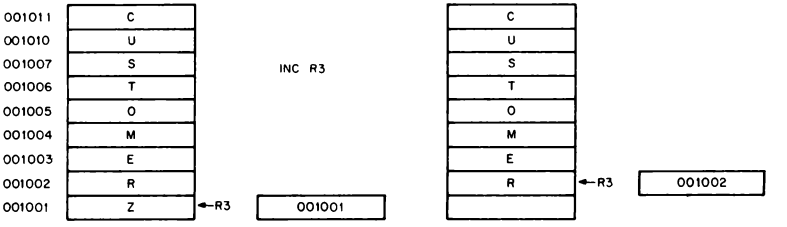


Figure 5-6: Byte Stack used as a Character Buffer

NOTE that in this case using the increment instruction (INC) is preferable to MOVB since it would accomplish the task of eliminating the unwanted character from the stack by readjusting the stack pointer without the need for a destination location. Also, the stack pointer (SP) used in this example cannot be the system stack pointer (R6) because R6 may only point to word (even) locations.

5.2 SUBROUTINE LINKAGE

5.2.1 Subroutine Calls

Subroutines provide a facility for maintaining a single copy of a given routine which can be used in a repetitive manner by other programs located anywhere else in memory. In order to provide this facility, generalized linkage methods must be established for the purpose of control transfer and information exchange between subroutines and calling programs. The PDP-11 instruction set contains several useful instructions for this purpose.

PDP-11 subroutines are called by using the JSR instruction which has the following format.

a general register (R) for linkage ——┐
JSR R,SUBR
an entry location (SUBR) for the subroutine —┘

When a JSR is executed, the contents of the linkage register are saved on the system R6 stack as if a MOV reg,-(SP) had been performed. Then the same register is loaded with the memory address following the JSR instruction (the contents of the current PC) and a jump is made to the entry location specified.

Address	Assembler Syntax	Octal Code
001000	JSR R5,SUBR	004567
001002	index constant for SUBR	000064
.	.	.
.	.	.
001064	SUBR: MOV A,B	01nnmm

Figure 5-7: JSR using R5

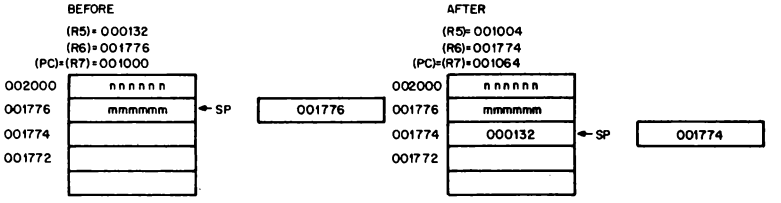


Figure 5-8: JSR

Note that the instruction JSR R6,SUBR is not normally considered to be a meaningful combination.

5.2.2 Argument Transmission

The memory location pointed to by the linkage register of the JSR instruction may contain arguments or addressses of arguments. These arguments may be accessed from the subroutine in several ways. Using Register 5 as the linkage register, the first argument could be obtained by using the addressing modes indicated by (R5), (R5)+,X(R5) for actual data, or @(R5)+, etc. for the address of data. If the autoincrement mode is used, the linkage register is automatically updated to point to the next argument.

Figures 5-9 and 5-10 illustrate two possible methods of argument transmission.

```
Address Instructions and Data

010400          JSR R5,SUBR
010402          Index constant for SUBR      SUBROUTINE CALL
010404          arg #1                       ARGUMENTS
010406          arg #2
.               .
.               .
.               .
.               .
.               .
020306  SUBR:   MOV (R5)+,R1        ;get arg #1
020310          MOV (R5)+,R2        ;get arg #2 Retrieve Arguments
                                    from SUB
```

Figure 5-9; Argument Transmission -Register Autoincrement Mode

```
Address  Instructions and Data

010400   JSR R5,SUBR
010402   index constant for SUBR    SUBROUTINE CALL
010404   077722                     Address of Arg #1
010406   077724                     Address of Arg. #2
010410   077726                     Address of Arg. #3
.        .                          .
.        .                          .
.        .                          .
077722   Arg #1
077724   arg #2                     arguments
077726   arg #3
.        .                          .
.        .                          .
.        .                          .
020306   SUBR:   MOV @(R5)+,R1  ;get arg #1
020301           MOV @(R5)+,R2  ;get arg #2
```

Figure 5-10: Argument Transmission-Register Autoincrement Deferred Mode

Another method of transmitting arguments is to transmit only the address of the first item by placing this address in a general purpose register. It is not necessary to have the actual argument list in the same general area as the subroutine call. Thus a subroutine can be called to work on data located anywhere in memory. In fact, in many cases, the operations performed by the subroutine can be applied directly to the data located on or pointed to by a stack without the need to ever actually move this data into the subroutine area.

```
Calling Program: MOV    POINTER, R1
                 JSR    PC,SUBR

SUBROUTINE       ADD    (R1)+,(R1)    ;Add item #1 to item #2, place
                                      result in item #2, R1 points
                                      to item #2 now
                        etc.
                        or

                 ADD    (R1),2(R1)    ;Same effect as above except that
                                      R1 still points to item #1
                                                        etc.
```

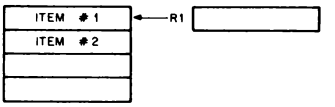


Figure 5-11: Transmitting Stacks as Arguments

Because the PDP-11 hardware already uses general purpose register R6 to point to a stack for saving and restoring PC and PS (processor status word) information, it is quite convenient to use this same stack to save and restore intermediate results and to transmit arguments to and from subroutines. Using R6 in this manner permits extreme flexibility in nesting subroutines and interrupt service routines.

Since arguments may be obtained from the stack by using some form of register indexed addressing, it is sometimes useful to save a temporary copy of R6 in some other register which has already been saved at the beginning of a subroutine. In the previous example R5 may be used to index the arguments while R6 is free to be incremented and decremented in the course of being used as a stack pointer. If R6 had been used directly as the base for indexing and not "copied", it might be difficult to keep track of the position in the argument list since the base of the stack would change with every autoincrement/decrement which occurs.

Figure 5-12: Shifting Indexed Base

However, if the contents of R6 (SP) are saved in R5 before any arguments are pushed onto the stack, the position relative to R5 would remain constant.

Figure 5-13: Constant Index Base Using "R6 Copy"

5.2.3 Subroutine Return

In order to provide for a return from a subroutine to the calling program an RTS instruction is executed by the subroutine. This instruction should specify the same register as the JSR used in the subroutine call. When executed, it causes the register specified to be moved to the PC and the top of the stack to be then placed in the register specified. Note that if an RTS PC is executed, it has the effect of returning to the address specified on the top of the stack.

Note that the JSR and the JMP Instructions differ in that a linkage register is always used with a JSR; there is no linkage register with a JMP and no way to return to the calling program.

When a subroutine finishes, it is necessary to "clean-up" the stack by eliminating or skipping over the subroutine arguments. One way this can be done is by insisting that the subroutine keep the number of arguments as its first stack item. Returns from subroutines would then involve calculating the amount by which to reset the stack pointer, resetting the stack pointer, then restoring the original contents of the register which was used as the copy of the stack pointer. The PDP-11/45, however, has a much faster and simpler method of performing these tasks. The MARK instruction which is stored on a stack in place of "number of argument" information may be used to automatically perform these "clean-up" chores. (For more information on the MARK instruction refer to Chapter 4.)

5.2.4 PDP-11 Subroutine Advantages

There are several advantages to the PDP-11 subroutine calling procedure.

a. arguments can be quickly passed between the calling program and the subroutine.

b. if the user has no arguments or the arguments are in a general register or on the stack the JSR PC,DST mode can be used so that none of the general purpose registers are taken up for linkage.

c. many JSR's can be executed without the need to provide any saving procedure for the linkage information since all linkage information is automatically pushed onto the stack in sequential order. Returns can simply be made by automatically popping this information from the stack in the opposite order of the JSR's.

Such linkage address bookkeeping is called automatic "nesting" of subroutine calls. This feature enables the programmer to construct fast, efficient linkages in a simple, flexible manner. It even permits a routine to call itself in those cases where this is meaningful (e.g. SQRT in FORTRAN SQRT(SQRT(X)). Other ramifications will appear after we examine the PDP-11 interrrupt procedures.

5.3 INTERRUPTS

5.3.1 General Principles

Interrupts are in many respects very similar to subroutine calls. However, they are forced, rather than controlled, transfers of program execution occurring because of some external and program-independent event (such as a stroke on the teleprinter keyboard). Like subroutines, interrupts have linkage information such

that a return to the interrupted program can be made. More information is actually necessary for an interrupt transfer than a subroutine transfer because of the random nature of interrupts. The complete machine state of the program immediately prior to the occurrence of the interrupt must be preserved in order to return to the program without any noticeable effects. (i.e. was the previous operation zero or negative, etc.) This information is stored in the Processor Status Word (PS). Upon interrupt, the contents of the Program Counter (PC) (address of next instruction) and the PS are automatically pushed onto the R6 system stack. The effect is the same as if:

```
MOV PS  ,-(SP)          ; Push PS
MOV R7,-(SP)            ; Push PC
```

had been executed.

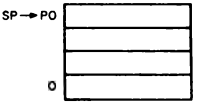
The new contents of the PC and PS are loaded from two preassigned consecutive memory locations which are called an "interrupt vector". The actual locations are chosen by the device interface designer and are located in low memory addresses of Kernel virtual space (see interrupt vector list, Appendix C). The first word contains the interrupt service routine address (the address of the new program sequence) and the second word contains the new PS which will determine the machine status including the operational mode and register set to be used by the interrupt service routine. The contents of the interrupt service vector are set under program control.

After the interrupt service routine has been completed, an RTI (return from interrupt) is performed. The two top words of the stack are automatically "popped" and placed in the PC and PS respectively, thus resuming the interrupted program.

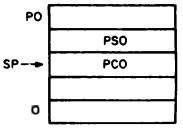
5.3.2 Nesting

Interrupts can be nested in much the same manner that subroutines are nested. In fact, it is possible to nest any arbitrary mixture of subroutines and interrupts without any confusion. By using the RTI and RTS instructions, respectively, the proper returns are automatic.

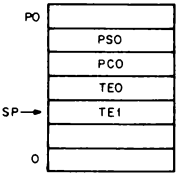
1. Process 0 is running; SP is pointing to location P0.



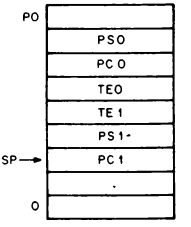
2. Interrupt stops process 0 with PC = PC0, and status = PS0 ;starts process 1.



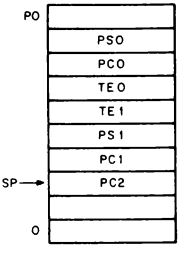
3. Process 1 uses stack for temporary storage (TE0, TE1).



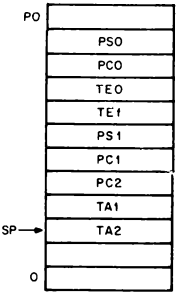
4. Process 1 interrupted with PC = PC1 and status = PS1; process 2 is started



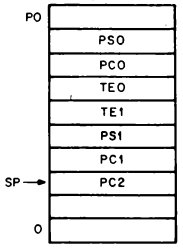
5. Process 2 is running and does a JSR R7,A to Subroutine A with PC = PC2.



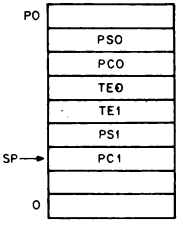
6. Subroutine A is running and uses stack for temporary storage.



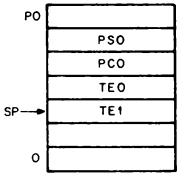
7. Subroutine A releases the temporary storage holding TA1 and TA2.



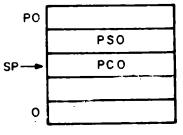
8. Subroutine A returns control to process 2 with an RTS R7,PC is reset to PC2.



9. Process 2 completes with an RTI instruction (dismisses interrupt) PC is reset to PC(1) and status is reset to PS1; process 1 resumes.



10. Process 1 releases the temporary storage holding TE0 and TE1.



11. Process 1 completes its operation with an RTI PC is reset to PC0 and status is reset to PS0.

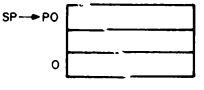


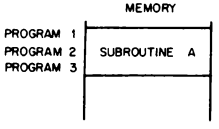
Figure 5-14: Nested Interrupt Service Routines and Subroutines

Note that the area of interrupt service programming is intimately involved with the concept of CPU and device priority levels. For a full discussion of the uses of the PDP-11/45 priority structure, refer to Chapter 2, System Architecture.

5.4 REENTRANCY

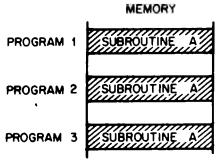
Further advantages of stack organization become apparent in complex situations which can arise in program systems that are engaged in the concurrent handling of several tasks. Such multi-task program environments may range from relatively simple single-user applications which must manage an intermix of I/O interrupt service and background computation to large complex multi-programming systems which manage a very intricate mixture of executive and multi-user programming situations. In all these applications there is a need for flexibility and time/memory economy. The use of the stack provides this economy and flexibility by providing a method for allowing many tasks to use a single copy of the same routine and a simple, unambiguous method for keeping track of complex program linkages.

The ability to share a single copy of a given program among users or tasks is called reentrancy. Reentrant program routines differ from ordinary subroutines in that it is unnecessary for reentrant routines to finish processing a given task before they can be used by another task. Multiple tasks can be in various stages of completion in the same routine at any time. Thus the following situation may occur:



PDP-11 Approach

Programs 1, 2, and 3 can share Subroutine A.



Conventional Approach

A separate copy of Subroutine A must be provided for each program.

Figure 5-15: Reentrant Routines

The chief programming distinction between a non-shareable routine and a reentrant routine is that the reentrant routine is composed solely of "pure code", i.e. it contains only instructions and constants. Thus, a section of program code is reentrant (shareable) if and only if it is "non self-modifying", that is it contains no information within it that is subject to modification.

Using reentrant routines, control of a given routine may be shared as illustrated in Figure 5-16.

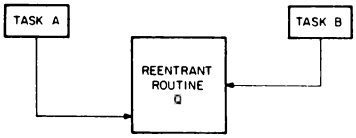


Figure 5-16: Reentrant Routine Sharing

1. Task A has requested processing by Reentrant Routine Q.
2. Task A temporarily relinquishes control (is interrupted) of Reentrant Routine Q before it finishes processing.
3. Task B starts processing in the same copy of Reentrant Routine Q.
4. Task B relinquishes control of Reentrant Routine Q at some point in its processing.
5. Task A regains control of Reentrant Routine Q and resumes processing from where it stopped.

The use of reentrant programming allows many tasks to share frequently used routines such as device interrupt service routines, ASCII-Binary conversion routines, etc. In fact, in a multi-user system it is possible for instance, to construct a reentrant FORTRAN compiler which can be used as a single copy by many user programs.

As an application of reentrant (shareable) code, consider a data processing program which is interrupted while executing a ASCII-to-Binary subroutine which has been written as a reentrant routine. The same conversion routine is used by the device service routine. When the device servicing is finished, a return from interrupt (RTI) is executed and execution for the processing program is then resumed where it left off inside the same ASCII-to-Binary subroutine.

Shareable routines generally result in great memory saving. It is the hardware implemented stack facility of the PDP-11 that makes shareable or reentrant routines reasonable.

A subroutine may be reentered by a new task before its completion by the previous task as long as the new execution does not destroy any linkage information or intermediate results which belong to the previous programs. This usually amounts to saving the contents of any general purpose registers, to be used and restoring them upon exit. The choice of whether to save and restore this information in the calling program or the subroutine is quite arbitrary and depends on the particular application. For example in controlled transfer situations (i.e. JSR's) a main program which calls a code-conversion utility might save the contents of registers which it needs and restore them after it has regained control, or the code conversion routine might save the contents of registers which it uses and restore them upon its completion. In the case of interrupt service routines this save/restore process must be carried out by the service routine itself since the interrupted program has no warning of an impending interrupt. The advantage of

using the stack to save and restore (i.e. "push" and "pop") this information is that it permits a program to isolate its instructions and data and thus maintain its reentrancy.

In the case of a reentrant program which is used to in a multi-programming environment it is usually necessary to maintain a separate R6 stack for each user although each such stack would be shared by all the tasks of a given user. For example, if a reentrant FORTRAN compiler is to be shared between many users, each time the user is changed, R6 would be set to point to a new user's stack area as illustrated in Figure 5-17.

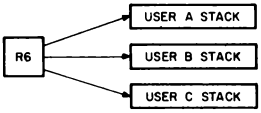


Figure 5-17: Multiple R6 Stack

5.5 POSITION INDEPENDENT CODE - PIC

Most programs are written with some direct references to specific addresses, if only as an offset from an absolute address origin. When it is desired to relocate these programs in memory, it is necessary to change the address references and/or the origin assignments. Such programs are constrained to a specifiec set of locations. However, the PDP-11 architecture permits programs to be constructed such that they are not constrained to specific locations. These Position Independent programs do not directly reference any absolute locations in memory. Instead all references are "PC-relative" i.e. locations are referenced in terms of offsets from the current location (offsets from the current value of the Program Counter (PC)). When such a program has been translated to machine code it will form a program module which can be loaded anywhere in memory as required.

Position Independent Code is exceedingly valuable for those utility routines which may be disk-resident and are subject to loading in a dynamically changing program environment. The supervisory program may load them anywhere it determines without the need for any relocation parameters since all items remain in the same positions relative to each other (and thus also to the PC).

Linkages to program routines which have been written in position independent code (PIC) must still be absolute in some manner. Since these routines can be located anywhere in memory there must be some fixed or readily locatable linkage addresses to facilitate access to these routines. This linkage address may be a simple pointer located at a fixed address or it may be a complex vector composed of numerous linkage information items.

5.6 RECURSION

It is often meaningful for a program routine to call itself as in the case of calculating a fourth root in FORTRAN with the expression SQRT(SQRT(X)). The ability to nest subroutine calls to the same subroutine is called recursion. The use of stack organization permits easy unambiguous recursion. The technique of recursion is of great use to the mathematical analyst as it also permits the evaluation of some otherwise non-computable mathematical functions. Although it is beyond the scope of this chapter to discuss the concept of recursive routines in detail, the reader should realize that this technique often permits very significant memory and speed economies in the linguistic operations of compilers and other higher-level software programs.

5.7 CO-ROUTINES

In some situations it happens that two program routines are highly interactive. Using a special case of the JSR instruction i.e. JSR PC,@(R6)+ which exchanges the top element of the Register 6 processor stack and the contents of the Program Counter (PC), two routines may be permitted to swap program control and resume operation where they stopped, when recalled. Such routines are called "co-routines". This control swapping is illustrated in Figure 5-18.

Routine #1 is operating, it then executes:

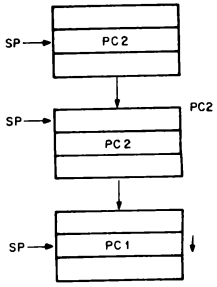
```
MOV #PC2,-(R6)

JSR PC,@(R6)+
```

with the following results:

1) PC2 is popped from the stack and the SP autoincremented

2) SP is autodecremented and the old PC (i.e. PC1) is pushed

3) control is transferred to the location PC2 (i.e. routine #2)



Routine #2 is operating, it then executes:

```
JSR PC ,@(R6)+
```

with the result the PC2 is exchanged for PC1 on the stack and control is transferred back to routine #1.

Figure 5-18 - Co-Routine Interaction

CHAPTER 6

MEMORY MANAGEMENT

The PDP-11/45 Memory Segmentation Unit provides the hardware facilities necessary for complete memory management and protection. It is designed to be a memory management facility for systems where the system memory size is greater than 28K words and for multi-user, multi-programming systems where memory protection and relocation facilities are necessary.

In order to most effectively utilize the power and efficiency of the PDP-11/45 in medium and large scale systems it is necessary to run several programs simultaneously. In such multi-programming environments several user programs would be resident in memory at any given time. The task of the supervisory program would be: control the execution of the various user programs, manage the allocation of memory and peripheral device resources, and safeguard the integrity of the system as a whole by careful control of each user program.

In a multi-programming system, the Segmentation Unit provides the means for assigning memory segments to a user program and preventing that user from making any unauthorized access to those segments outside his assigned area. Thus, a user can effectively be prevented from accidental or willful destruction of any other user program or the system executive program.

The basic characteristics of the PDP-11/45 Memory Segmentation Unit are:

- 16 User mode memory segments
- 16 Supervisor mode memory segments
- 16 Kernel mode memory segments
- 8 segments in each mode for instructions
- 8 segments in each mode for data
- segment lengths from 32 to 4096 words
- each segment provided with full protection and relocation
- transparent operation
- 6 modes of memory access control
- memory extension to 124K words (248K bytes)

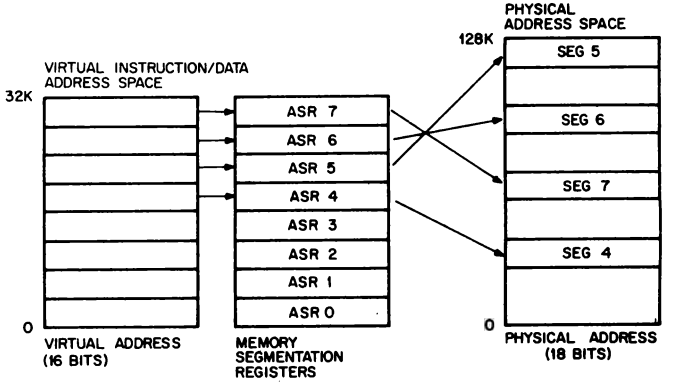
6.1 PDP-11 FAMILY BASIC ADDRESSING LOGIC

The addresses generated by all PDP-11 Family Central Processor Units (CPUs) are 18-bit direct byte addresses. Although the PDP-11 Family word length and operational logic is all 16-bit length, the UNIBUS and CPU addressing logic actually is 18-bit length. Thus, while the PDP-11 word can only contain address references up to 32K words (64K bytes) the CPU and UNIBUS can reference addresses up to 128K words (256K bytes). These extra two bits of addressing logic provide the basic framework for expanded memory segmentation.

In addition to the word length constraint on basic memory addressing space, the uppermost 4K words of address space is always reserved for UNIBUS I/O device registers. In a basic PDP-11/45 memory configuration (without Segmentation) all address references to the uppermost 4K words of 16 bit address space (170000-177777) are converted to full 18-bit references with bits 17 and 16 always set to 1. Thus, a 16 bit reference to the I/O device register at address 173224 is automatically internally converted to a full 18-bit reference to the register at address 773224. Accordingly, the basic PDP-11/45 configuration can directly address up to 28K words of true memory, and 4K words of UNIBUS I/O device registers. Memory configurations beyond this require the PDP-11/45 Memory Segmentation Unit.

6.2 VIRTUAL ADDRESSING

When the PDP-11/45 Memory Segmentation Unit is operating, the normal 16 bit direct byte address is no longer interpreted as a direct Physical Address (PA) but as a Virtual Address (VA) containing information to be used in constructing a new 18-bit physical address. The information contained in the Virtual Address (VA) is combined with relocation and description information contained in the Active Segment Register (ASR) (see 6.4) to yield an 18-bit Physical Address (PA). Using the Segmentation Unit, memory can be dynamically allocated in segments each composed of from 1 to 128 integral blocks of 32 words.



ASR = Active Segment Register

Figure 6-1 Virtual Address Mapping into Physical Address

The starting address for each segment is an integral multiple of 32 words, and each segment has a maximum size of 4096 words. Segments may be located anywhere within the 128K Physical Address space. The determination of which set of 16 segmentation registers is used to form a Physical Address is made by the current mode of operation of the CPU, i.e. Kernel, Supervisor or User mode.

6.3 INTERRUPT CONDITIONS UNDER SEGMENTATION CONTROL

The Memory Segmentation Unit relocates all addresses. Thus, when Segmentation is enabled, all trap, abort, and interrupt vectors are considered to be in Kernel mode Virtual Address Space. When a vectored transfer occurs, control is transferred according to a new Program Counter (PC) and Processor Status Word (PS) contained in a two-word vector relocated through the Kernel Active Segment Register Set.

When a trap, abort, or interrupt occurs the "push" of the old PC, old PS is to the User/Supervisor/Kernel R6 stack specified by CPU mode bits 15,14 of the new PS in the vector (bits 15,14: 00 = Kernel, 01 = Supervisor, 11 = User). The CPU mode bits also determine the new ASR set. In this manner it is possible for a Kernel mode program to have complete control over service assignments for all interrupt conditions, since the interrupt vector is located in Kernel space. The Kernel program may assign the service of some of these conditions to an Supervisor or User mode program by simply setting the CPU mode bits of the new PS in the vector to return control to the appropriate mode.

6.4 CONSTRUCTION OF A PHYSICAL ADDRESS

All addresses with memory segmentation either reference information in Instruction (I) Space or Data (D) Space. I Space is used for all instruction fetches, index words, absolute addresses and immediate operands, D Space is used for all other references. I Space and D Space each have 8 ASR's in each mode of CPU operation, Kernel, Supervisor, and User. Using Segment Status Register #3 (6.6.4), the operating system may select to disable D space and map all references (Instructions and Data) through I space, or to use both I and D space.

The basic information needed for the construction of a Physical Address (PA) comes from the Virtual Address (VA). which is illustrated in Figure 6-2, and the appropriate ASR set.

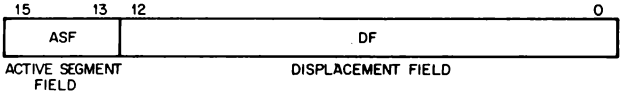


Figure 6-2: Interpretation of a Virtual Address

The Virtual Address (VA) consists of:

1. The Active Segment Field (ASF). This 3-bit field determines which of eight Active Segment Registers (ASR0-ASR7) will be used to form the Physical Address (PA).

2. The Displacement Field (DF). This 13-bit field contains an address relative to the beginning of a segment. This permits segment lengths up to 4K words (2_{13} = 8K bytes). The DF is further subdivided into two fields as shown in Figure 6-3.

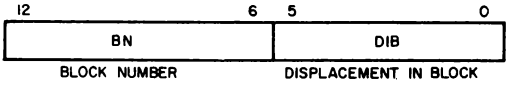


Figure 6-3: Displacement Field of Virtual Address

The Displacement Field (DF) consists of:

1. The Block Number (BN). This 7-bit field is interpreted as the block number within the current segment.

2. The Displacement in Block (DIB). This 6-bit field contains the displacement within the block referred to by the Block Number (BN).

The remainder of the information needed to construct the Physical Address comes from the 12-bit Segment Address Field (SAF) (part of the Active Segment Register (ASR)) and specifies the starting address of the memory segment which that ASR describes. The SAF is actually a block number in the physical memory, e.g. SAF = 3 indicates a starting address of 96(3 x 32) words in physical memory.

The formation of a physical address (PA) takes 90 ns. Thus in situations which do not require the facilities of the Memory Segmentation Unit, it should be disabled to permit time savings.

The formation of the Physical Address (PA) is illustrated in Figure 6-4.

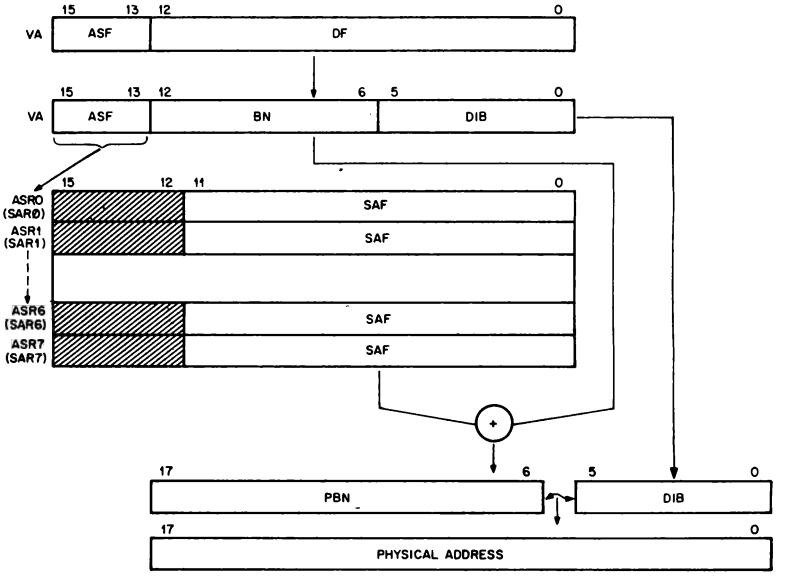


Figure 6-4: Construction of a Physical Address

The logical sequence involved in constructing a Physical Address (PA) is as follows:

1. Select a set of Active Segment Registers depending on the space being referenced.
2. The Active Segment Field (ASF) of the Virtual Address is used to select an Active Segment Register (ASR0-ASR7).
3. The Segment Address Field (SAF) of the selected Active Segment Register (ASR) contains the starting address of the currently active segment as a block number in physical memory.
4. The Block Number (BN) from the Virtual Address (VA) is added to the block number from the Segment Address Field (SAF) to yield the number of the block in physical memory (PBN-Physical Block Number) which will contain the Physical Address (PA) being constructed.
5. The Displacement in Block (DIB) from the Displacment Field (DF) of the Virtual Address (VA) is joined to the Physical Block Number (PBN) to yield a true 18-bit PDP-11/45 Physical Address (PA).

6.5 SEGMENTATION REGISTERS

The PDP-11/45 Memory Segmentation Unit implements three sets of sixteen 32-bit Active Segment Registers (ASR). Each set is subdivided into two sets of eight ASR's, one for references to Instruction (I) Space, and one to Data (D) Space. An Active Segment Register (ASR) is actually a pair of 16-bit registers: a Segment Address Register (SAR) and a Segment Descriptor Register (SDR). These registers are always used as a pair and contain all the information needed to describe and locate the currently active memory segments.

One set of Active Segment Registers (ASR) is used in Kernel mode, another in Supervisor, and the other in User mode. The choice of which set is to be used is determined by the current CPU mode contained in the Processor Status word.

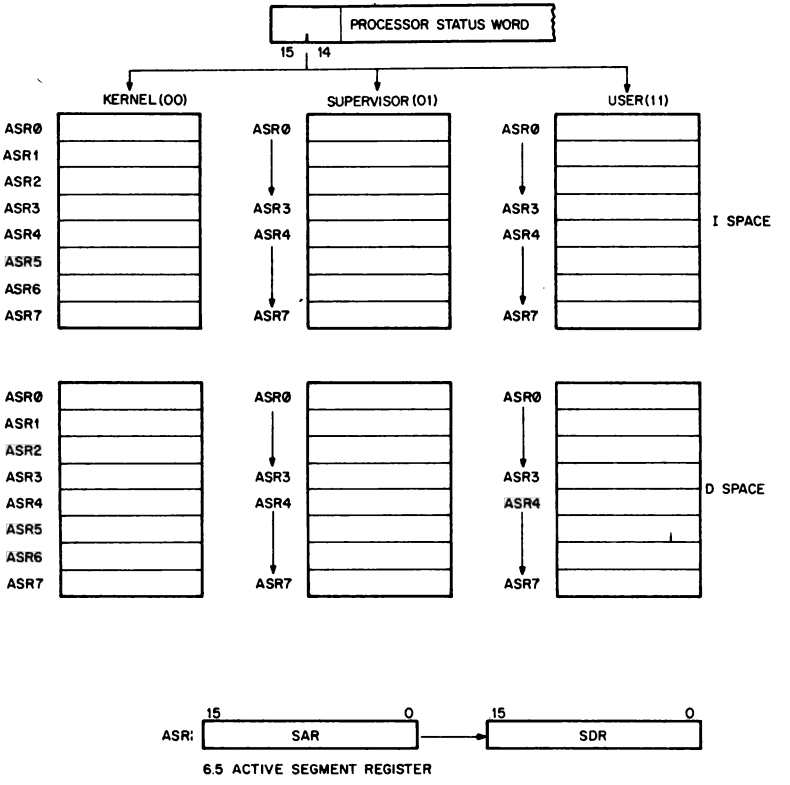


Figure 6-5: Active Segment Registers

The various Memory Segmentation Registers are located in the uppermost 4K of PDP-11 physical address space along with the UNIBUS I/O device registers. For the actual addresses of these registers refer to Paragraph 6.9, Memory Segmentation Unit - Register Map.

6.5.1 Segment Address Registers (SAR)

The Segment Address Register (SAR) is the first word of the 32-bit Active Segment Register (ASR); it contains the Segment Address Field (SAF), a 12-bit field, which specifies the starting address of the segment as a block number in physical memory.

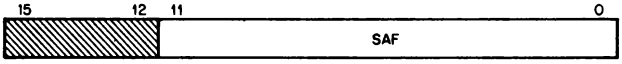


Figure 6-6: Segment Address Register

Bits 15-12 of the SAR are unused and reserved for possible future use.

The Segment Address Register (SAR) which contains the Segment Address Field (SAF) may be alternatively thought of as a relocation register containing a relocation constant, or as a base register containing a base address. Either interpretation indicates the basic importance of the Segment Address Register (SAR) as a relocation tool.

6.5.2 Segment Descriptor Register

The Segment Descriptor Register (SDR) contains information relative to segment expansion, segment length, and access control.

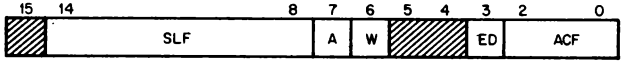


Figure 6-7: Segment Descriptor Register

6.5.2.1 Access Control Field (ACF)

This three-bit field, occupying bits 2-0 of the Segment Descriptor Register (SDR) contains the access rights to this particular segment. The access codes or "keys" specify the manner in which a segment may be accessed and whether or not a given access should result in a trap or an abort of the current operation. A memory reference which causes an abort is not completed while a reference causing a trap is completed. In fact, when a memory reference causes a trap to occur, the trap does not occur until the entire instruction has been completed. Aborts are used to catch "missing segment faults", prevent illegal access, etc.; traps are used as an aid in gathering memory management information.

In the context of access control the term "write" is used to indicate the action of any instruction which modifies the contents of any addressable word. Except in those cases where references are made to the 4K word UNIBUS I/O register area, a "write" is synonomous with what is usually called a "store" or "modify" in many computer systems.

The modes of access control are as follows:

000	non-resident	abort all accesses
100	read/write	memory management trap upon completion of a read or write
101	read/write	memory management trap upon completion of a write
110	read/write	no system trap/abort action
001	read-only	abort on write attempt memory management trap on read
010	read-only	abort on write attempt
011	unused	abort all accesses - reserved for future use
111	unused	abort all accesses - reserved for future use

It should be noted that the use of I Space provides the user with a further form of protection, execute only.

6.5.2.2 Access Information Bits
A Bit (bit 7) - This bit is used by software to determine whether or not any accesses to this segment met the trap condition specified by the Access Control Field (ACF). (A = 1 is Affirmative) The A Bit is used in the process of gathering memory mangement statistics.

W Bit (bit 6) - This bit indicates whether or not this segment has been modified (i.e. written into) since the ASR was loaded. (W = 1 is Affirmative) The W Bit is useful in applications which involve disk swapping and memory overlays. It is used to determine which segments have been modified and hence must be saved in their new form and which segments have not been modified and can be simply overlayed.

Note that A and W bits are "reset" to "0" whenever the Active Segment Register (ASR) (either SAR or SDR) is modified (written into).

6.5.2.3 Expansion Direction (ED)
This one-bit field, located at bit 3 of the Segment Descriptor Register (SDR), specifies whether the segment expands upward from relative zero (ED = 0) or downwards toward relative zero (ED = 1). Relative zero, in this case, is the SAF (Segment Address Field). Expansion is done by changing the Segment Length Field (6.5.2.4). In expanding upwards, blocks with higher relative addresses are added; in expanding downwards, blocks with lower relative addresses are added to the segment. Upward expansion is usually used to add more program space, while downward expansion is used to add more stack space.

6.5.2.4 Segment Length Field (SLF)

The seven-bit field, occupying bits 14-8 of the Segment Descriptor Register (SDR), specifies the number of blocks in the segment. A segment consists of at least one and at most 128 blocks, and occupies contiguous core locations. If the segment expands upwards, this field contains the length of the segment minus one (in blocks). If the segment expands downwards, this field contains 128 minus the length of the segment (in blocks).

A Segment Length Error occurs when the Block Number (BN) of the virtual address (VA) is greater than the Segment Length Field (SLF), if the segment expands upwards, or if the segment expands downwards, when the BN is less than the SLF.

6.5.2.5 Reserved Bits

Bits 15, 4 and 5 are reserved for future use, and are always 0.

6.6 SEGMENTATION FAULT RECOVERY REGISTERS

Aborts and traps generated by the segmentation hardware are vectored through Kernel virtual location 250, Segmentation Status Registers #0, #1 and #2 are used in order to differentiate an abort from a trap, determine why the abort or trap occurred, and allow for easy program restarting. Note that an abort or trap to a location which is itself an invalid address will cause another abort or trap. Thus the Kernel program must insure that Kernel Virtual Address 10 is mapped into a valid address, otherwise a loop will occur which will require console intervention.

6.6.1 Segment Status Register #0 (SSR0) (segmentation status and error indicators)

SSR0 contains error flags, the segment number whose reference caused the abort, and various other status flags. The register is organized as shown in Figure 6-8.

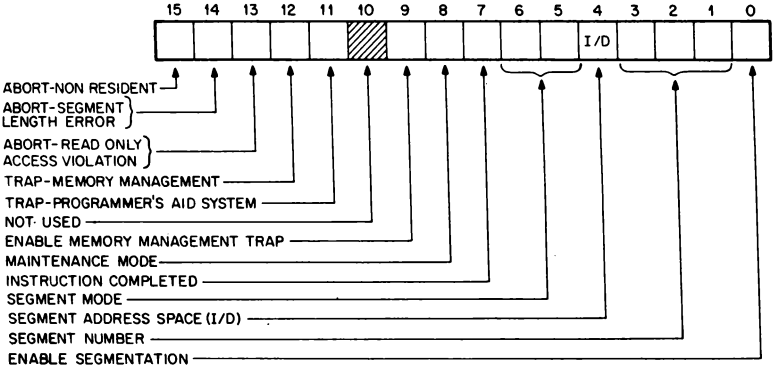


Figure 6-8: Format of Segmentation Status Register #0 (SSR0)

Bits 15-11 are the error flags. They may be considered to be in a "priority queue" in that "flags to the right" are less significant and should be ignored. That is, a "non-resident" fault service routine would ignore segment length, access control, and memory management flags. A "segment length" service routine would ignore access control and memory management faults, etc.

Bits 15-13 when set (error conditions) cause Memory Segmentation to freeze the contents of bits 1-7 and Segment Status Registers #1 and #2. This has been done to facilitate error recovery (discussed in 6.6.3).

Bits 15-12 are enabled by a signal called "SEG" . "SEG" is true when an address is being relocated by the segmentation unit. This implies that either SSR0, bit 0 is equal to 1 (segmentation operating) or that SSR0, bit 8, (MAINTENANCE) is equal to 1 and the memory reference is the final one of a destination calculation (maintenance/destination mode).

Note that Segmentation Status Register #0 (SSR0) bits 0, 8, and 9 can be set under program control to provide meaningful segmentation control information. However, information written into all other bits is not meaningful. Only that information which is automatically written into these remaining bits as a result of hard ware actions is useful as a monitor of the status of the Memory Segmentation Unit. Setting bits 15-11 under program control will not cause traps to occur; these bits however must be reset to 0 after an abort or trap has occurred in order to resume segmentation status monitoring.

6.6.1.1 Abort - Non-Resident

Bit 15 is the "Abort - Non-Resident" bit. It is set by attempting to access a segment with an Access Code Field (ACF) key equal to 0, 3, or 7. It is also set by attempting to use Memory Segmentation with a mode of 2.

6.6.1.2 Abort - Segment Length

Bit 14 is the "Abort Segment Length" bit. It is set by attempting to access a location in a segment with a block number (Virtual Address bits 12-6) that is outside the area authorized by the Segment Length Field (SLF) of the Active Segment Register (ASR) for that segment. Bits 14 and 15 may be set simultaneously by the same access attempt.

6.6.1.3 Abort - Read Only

Bit 13 is the "Abort - Read-Only" bit. It is set by attempting to write in a "Read-Only" segment. "Read-Only" segments have access keys of 1 or 2.

6.6.1.4 Trap - Memory Management

Bit 12 is the "Trap-Memory Management" bit. It is set by a read operation which references segment with an Access Control Field (ACF) of 1 or 4, or by a write operation to a segment with an ACF key of 4 or 5.

6.6.1.5 Trap - Programmer's Aid System Flag

Bit 11 is the "Trap-Programmer's Aid System" bit. It may be set independently of whether the Memory Sementation Unit option is enabled or not. It is used with the Programmer's Aid System (PAS) (a separate option which provides hardware "break-point" facilities) to cause a trap on a specific address or data match. The PAS "trap conditions" can be set by either program or PAS console switch control. When these "trap conditions" are met, a trap, vectored through Kernel Virtual Address 10 is generated upon completion of the current instruction. The service routine must interrogate SSR0 to determine exactly which "trap condition" caused the trap to occur.

6.6.1.6 Bit 10
Bit 10 is a spare location and is always equal to 0. It is unused and reserved for possible future expansion.

6.6.1.7 Enable Memory Management Traps
Bit 9 is the "Enable Memory Management Traps" bit. It can be set or cleared by doing a direct write into SSR0. If bit 9 is 0, no Memory Managment traps will occur. The A and W bits will, however, continue to log potential Memory Management Traps. When bit 9 is set to 1, the next "potential" Memory Managment trap will cause a trap, vectored through Kernel Virtual Address 10.

Note that if an instruction which sets bit 9 to 0 (disable Memory Management Trap) causes a potential Memory Management trap in the course of any of its memory references prior to the one actually changing SSR0, then the trap will occur at the end of the instruction anyway.

Note also that bit 9 has no effect on Programmer's Aid System (PAS) traps.

6.6.1.8 Maintenance/Destination Mode
Bit 8 specifies Maintenance use of the Memory Segmentation Unit. It is provided for diagnostic purposes only.

6.6.1.9 Instruction Completed
Bit 7 indicates that the current instuction has been completed. It will be set to 0 whenever an instruction is aborted. This provides error handling routines with a way of determining whether the last instruction will have to be repeated in the course of an error recovery attempt. Bit 7 is Read-Only (it cannot be written). It is initialized to a 1.

6.6.1.10 Segment Mode
Bits 5, 6 indicate the CPU mode (User/Supervisor/Kernel) associated with the segment causing the abort. (Kernel = 00, Supervisor = 01, User = 11) If an illegal mode (10) is specified, segmentation will abort and set bit 15.

6.6.1.1.11 Segment Address Space
Bit 4 indicates the type of address space (I or D) the Segmentation Unit was in when a segmentation fault occurred (0 = I Space, 1 = D Space). It is used in conjunction with bits 3-1, Segment Number.

6.6.1.12 Segment Number
Bits 3-1 contain the segment number of a reference causing a segmentation fault. Note that segments, like blocks, are numbered from 0 upwards.

6.6.1.13 Enable Segmentation
Bit 0 is the "Enable Segmentation" bit. When it is set to 1, all addresses are relocated by the Segmentation unit. When bit 0 is set to 0 the Memory Segmentation Unit is inoperative and addresses are not relocated or protected.

6.6.2 Segmentation Status Register # 1 (SSR1)
SSR1 records any autoincrement/decrement of the general purpose registers, including explicit references through the PC. SSR1 is cleared at the beginning of each instruction fetch. Whenever a general purpose register is either autoincremented or autodecremented the register number and the amount (in 2s complement notation) by which the register was modified, is written into SSR1.

The information contained in SSR1 is necessary to accomplish an effective recovery from an error resulting in an abort. The low order byte is written first and it is not possible for a PDP-11 instruction to autoincrement/decrement more than two general purpose registers per instruction before an "abort-causing" reference. Register numbers are recorded "MOD 8"; thus it is up to the software to determine which set of registers (User/Supervisor/Kernel--General Set 0/General Set 1) was modified, by determining the CPU and Register modes as contained in the PS at the time of the abort. The 6-bit displacement on R6(SP) that can be caused by the MARK instruction cannot occur if the instruction is aborted.

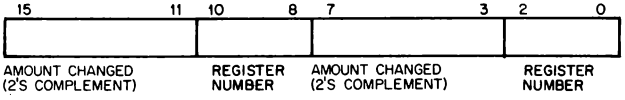


Figure 6-9: Format of Segmentation Status Register #1 (SSR)

6.6.3 Segment Status Register #2

SSR2 is loaded with the 16-bit Virtual Address (VA) at the beginning of each instruction fetch, or with the address Trap Vector at the beginning of an interrupt, "T" Bit trap, Parity, Odd Address, and Timeout traps. SSR2 is Read-Only; it can not be written. SSR2 is the Virtual Address Program Counter.

6.6.4 Segment Status Register #3

The Segment Status Register #3 (SSR3) enables or disables the use of the D space ASR's. When D space is disabled, all references use the I space registers; when D space is enabled, both the I space and D space registers are used. Bit 0 refers to the User's Registers, Bit 1 to the Supervisor's, and Bit 2 to the Kernel's. When the appropriate bits are set D space is enabled; when clear, it is disabled. Bits 3-15 are unused. On initialization this register is set to 0 and only I space is in use.

6.6.5 Instruction Back-Up/Restart Recovery

The process of "backing-up" and restarting a partially completed instruction involves:

1. Performing the appropriate memory management tasks to alleviate the cause of the abort (e.g. loading a missing segment, etc.)
2. Restoring the general purpose registers indicated in SSR1 to their original contents at the start of the instruction by subtracting the "modify value" specified in SSR1
3. Restoring the PC to the "abort-time" PC by loading R7 with the contents of SSR2, which contains the value of the Virtual PC at the time the "abort-generating" instuction was fetched.

Note that this back-up/restart procedure assumes that the general purpose register used in the program segment will not be used by the abort recovery routine. This is the case if the aborted routine is in User mode while the recovery routine is in Kernel mode, since the recovery program will probably use a different general register set.

6.6.6 Clearing Segment Status Registers Following Trap/Abort

At the end of a segmentation fault service routine bits 15-11 of SSR0 must be cleared (set to 0) to resume segmentation error checking. On the next memory reference following the clearing of these bits, the various Segmentation Status Registers will resume monitoring the status of the segmentation addressing operations, (SSR2) will be loaded with the next instruction address, SSR1 will store register change information and SSR0 will log Memory Segmentation Status information).

6.7 SEGMENTATION EXAMPLES

6.7.1 Normal Usage

The Memory Segmentation Unit provides a very general purpose memory management tool. It can be used in a manner as simple or complete as desired. It can be anything from a simple memory expansion device to a very complete memory management facility.

The variety of possible and meaningful ways to utilize the facilities offered by the Memory Segmentation Unit means that both single-user and multi-programming systems have complete freedom to make whatever memory management decisions best suit their individual needs. Although a knowledge of what most types of computer systems seek to achieve may indicate that certain methods of utilizing the Memory Segmentation Unit will be more common than others, there is no limit to the ways to use these facilities.

In most normal applications, it is assumed that the control over the actual memory segment assignments and their protection resides in a supervisory type program which would operate at the nucleus of a CPU's executive (Kernel mode). It is further assumed that this Kernel mode program would set access keys in such a way as to protect itself from willful or accidental destruction by other Supervisor mode or User mode programs. The facilities are also provided such that this nucleus can dynamically assign memory segments of varying sizes in response to system needs.

6.7.2 Typical Memory Segment

When the Memory Segmentation Unit is enabled, the Kernel mode program, a Supervisor mode program and a User mode program each have eight active segments described by the appropriate Active Segment Registers (ASR) for data, and eight, for instructions. Each segment is made up of from 1 to 128 blocks and is pointed to by the Segment Address Field (SAF) of the corresponding Segment Address Register (SAR) as illustrated in Figure 6-10.

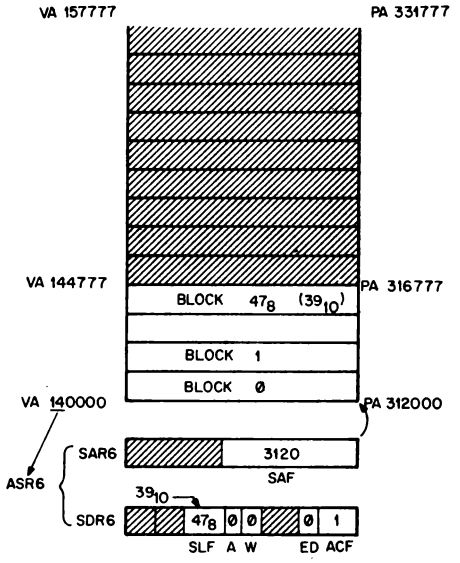


Figure 6-10: Typical Memory Segment

The memory segment illustrated in Figure 6-10 has the following attributes:

1. Segment Length: 40 blocks
2. Virtual Address Range: 140000 - 144777
3. Physical Address Range: 312000 - 316777
4. No trapped access has been made to this segment
5. Nothing has been modified (i.e. written) in this segment.
6. Read-Only Protection.
7. Upward Expansion

These attributes were determined according to the following scheme:

1. Active Segment Register ASR6 with its component Segment Address Register (SAR6) and Segment Descriptor Register (SDR6) were selected by the Active Segment Field (ASF) of the Virtual Address (VA). (Bits 15-13 of the VA = 6_8.)

2. The initial address of the segment was determined from the Segment Address Field (SAF) of ASR6 (312000 = 3120_x blocks x 40_x (32_{10}) words per block x 2 bytes per word).

   Note that the SAR which contains the SAF constitutes what is often referred to as a base register containing a base address or a relocation register containing relocation constant.

3. The segment length ($47_x + 1 = 40_{10}$ blocks) was determined from the Segment Length Field (SLF) contained in Segment Descriptor Register SDR6. Any attempts to reference beyond these 40_{10} blocks in this segment will cause a "Segment Length Error" ,which will result in an abort, vectored through Kernel Virtual Address 10.

4. The Physical Addresses were constructed according to the scheme illustrated in Figure 6-4.

5. The Access bit (A-bit) of SDR6 indicates that no trapped access has been made to this segment (A bit = 0). When an illegal or trapped reference, (i.e. a violation of the Protection Mode specified by the Access Control Field (ACF) for this segment), or a trapped reference (i.e. Read in this case), occurs, the A-bit will be set to a 1.

6. The Written bit (W-bit) indicates that no locations in this segment have been modified (i.e.written). If an attempt is made to modify any location in this particular segment, an Access Control Violation Abort will occur. If this segment were involved in a disk swapping or memory overlay scheme, the W-bit would be used to determine whether it had been modified and thus required saving before overlay.

7. This segment is Read-Only protected; i.e. no locations in this segment may be modified. In addition, a memory management trap will occur upon completion of a read access. The mode of protection was specified by the Access Control Field (ACF) of SDR6.

8. The direction of expansion is upward (ED = 0). If more blocks are required in this segment, they will be added by assigning blocks with higher relative addresses.

Note that the various attributes which describe this segment can all be determined under software control. The parameters describing the segment are all loaded into the appropriate Active Segment Register (ASR) with its component Segment Address Register (SAR) and Segment Descriptor Register (SDR) under program control. In a normal application it is assumed that the particular segment which itself contains these registers would be assigned to the control of a supervisory type program operating in Kernel mode.

6.7.3 Non-Consecutive Memory Segments

It should be noted at this point that although the correspondence between Virtual Addresses (VA) and Active Segment Register (ASR) is such that higher VAs have higher ASRs, this does not mean that higher Virtual Addresses (VA) necessarily correspond to higher Physical Addresses (PA). It is quite simple to set up the Segment Address Fields (SAF) of the ASRs in such a way that higher Virtual Address blocks may be located in lower Physical Address blocks as illustrated in Figure 6-11.

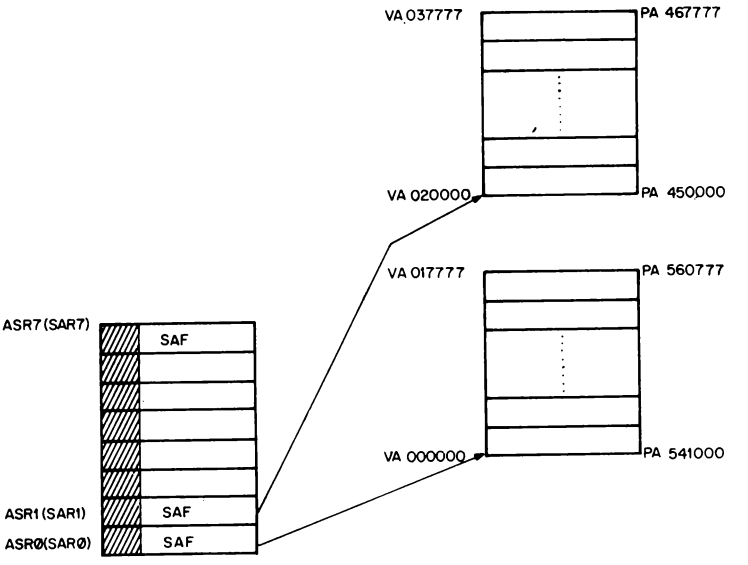


Figure 6-11: Non-Consecutive Memory Segments

Note that although a single memory segment must consist of a block of contiguous locations, memory segments as macro units do not have to be located in consecutive Physical Address (PA) location. It also should be realized that the assignment of memory segments is not limited to consecutive non-overlapping Physical Address (PA) locations.

6.7.4 Stack Memory Segments

When constructing PDP-11/45 programs it is often desirable to isolate all program variables from "pure code" (i.e. program instructions) by placing them on a register indexed stack. These variables can then be "pushed" or "popped" from the stack area as needed (see Chapter 3, Addressing Modes). Since all PDP-11 Family stacks expand by adding locations with lower addresses, when a memory segment which contains "stacked" variables needs more room it must "expand down", i.e. add blocks with lower relative addresses to the current segment. This mode of expansion is specified by setting the Expansion Direction (ED) bit of the appropriate Segment Descriptor Register (SDR) to a 1. Figure 6-12 illustrates a typical "stack" memory segment. This segment will have the following parameters:

ASR6: SAF = 3120

SDR6: SLF = 175_8 or 125_{10} (128_{10}-3)

ED = 1

A = 0 or 1

W = 0 or 1

ACF = nnn (to be determined by programmer as the need dictates).

note: the A, W bits will normally be set by hardware

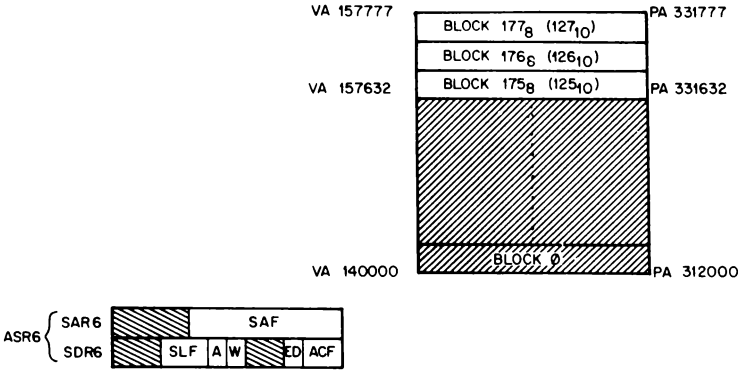


Figure 6-12: Typical Stack Memory Segment

In this case the stack begins 128 blocks above the relative origin of this memory segment and extends downward for a length of three blocks. A "SEGMENT LENGTH ERROR" abort vectored through Kernel Virtual Address (VA) 10 will be generated by the segmentation hardware when an attempt is made to reference any location below the assigned area, i.e. when the Block Number (BN) from the Virtual Address (VA) is less than the Segment Length Field (SLF) of the appropriate Active Segment Register (ASR).

6.8 TRANSPARENCY

It should be clear at this point that in a multiprogramming application it is possible for memory segments to be allocated in such a way that a particular program seems to have a complete 32K basic PDP-11/45 memory configuration. Using Memory Segmentation, a Kernel Mode supervisory-type program can easily perform all memory managment tasks in a manner entirely transparent to a Supervisor or User mode program. In effect, a PDP-11/45 System can utilize its resources to provide maximum through-put and response to a variety of users each of which seems to have a powerful system "all to himself" .

6.9 MEMORY SEGMENTATION UNIT - REGISTER MAP

REGISTER	ADDRESS
Segmentation Status Register # 0(SSR0)	777572
Segmentation Status Register # 1(SSR1)	777574
Segmentation Status Register #.2(SSR2)	777576
Segmentation Status Register# 3(SSR3)	772516
User I Space Descriptor Register (UISDR0)	777600
.	.
.	.
.	.
.	
User I Space Descriptor Register (UISDR7)	777616
User D Space Descriptor Register (UDSDR0)	777620
.	.
.	.
.	.
.	
User D Space Descriptor Register (UDSDR7)	777636
User I Space Address Register (UISAR0)	777640
.	.
.	.
.	.
User I Space Address Register (UISAR7)	777656
User D Space Address Register (UDSAR0)	777660
.	.
.	.
.	.
User D Space Address Register (UDSAR7)	777676
Supervisor I Space Descriptor Register (SISDR0)	772200
.	.
.	.
.	.
Supervisor I Space Descriptor Register (SISDR7)	772216
Supervisor D Space Descriptor Register (SDSDR0)	772226
.	.
.	.
.	.
Supervisor D Space Descriptor Register (SDSDR7)	772236
Supervisor I Space Address Register (SISAR0)	772240
.	.
.	.
.	.
Supervisor I Space Address Register (SISAR7)	772256

Supervisor D Space Address Register (SDSAR0)	772260
.	.
.	.
.	.
Supervisor D Space Address Register (SDSDR7)	772276
Kernel I Space Descriptor Register (KISDR0)	772300
.	.
.	.
.	.
Kernel I Space Descriptor Register (KIDSR7)	772316
Kernel D Space Descriptor Register (KDSDR0)	772320
.	.
.	.
.	.
Kernel D Space Descriptor Register (KDSDR7)	772336
Kernel I Space Address Register (KISAR0)	772340
.	.
.	.
.	.
Kernel I Space Address Register (KISAR7)	772356
Kernel D Space Address Register (KDSAR0)	772360
.	.
.	.
.	.
Kernel D Space Address Register (KDSAR7)	772376

CHAPTER 7

FLOATING POINT PROCESSOR

7.1 INTRODUCTION

The PDP-11 Floating Point Processor is an optional arithmetic processor which fits integrally into the PDP-11/45 Central Processor. It performs all floating point arithmetic operations and converts data between integer and floating point formats.

The hardware provides a time and money saving alternative to the use of software floating point routines. Its use can result in many orders of magnitude improvement in the execution of arithmetic operations.

The features of the unit are:

- Overlapped operation with central processor
- High speed
- Single and double precision (32 or 64 bit) floating point modes
- Flexible addressing modes
- Six 64-bit floating point accumulators
- Error recovery aids.

7.2 OPERATION

The Floating Point Processor is an integral part of the Central Processor. It operates using similar address modes, and the same memory management facilities provided by the Memory Segmentation Option, as the Central Processor. Floating Point Processor instructions can reference the floating point accumulators, the Central Processor's general registers, or any location in memory.

When, in the course of a program, an FPP Instruction is fetched from memory, the FPP will execute that instruction in parallel with the CPU continuing with its instruction sequence. The CPU is delayed a very short period of time during the FPP instruction's Fetch operation, and then is free to proceed independently of the FPP. The interaction between the two processors is automatic, and a program can take full advantage of the parallel operation of the two processors by intermixing Floating Point Processor and Central Processor instructions.

Interaction between Floating Point Processor and Central Processor instructions is automatically taken care of by the hardware. When an FPP instruction is encountered in a program, the machine first checks the status of the Floating Point Processor. If the FPP is "busy", the CPU will wait until it is "done"before continuing execution of the program.

```
        LDD (R3)+,AC3       ;Pick up constant operand and place it in
                            AC3.
ADDLP:  LDD (R3)+,AC0       ;Load AC0 with next value in table
        MULD AC3,AC0        ;and multiply by constant in AC3
        ADDD AC0,AC1        ;and add the result into AC1
        SOB R5,ADDLP        ;check to see whether done
        STCDI AC1,@R4       ;done, convert double to integer and store
```

In the above example the Floating Point Processor would execute the next three instructions. After the "ADDD"was fetched into the FPP, the CPU would execute the "SOB"and then wait for the FPP to be "done"with the "ADDD"before giving it the "LDD"or "STCDI"instruction.

As can be seen from this example, autoincrement and autodecrement addressing automatically adds or subtracts the correct amount to the contents of the register depending on the modes represented by the instruction.

7.3 ARCHITECTURE

The Floating Point Processor contains scratch registers, a Floating Exception Address pointer (FEA), a Program Counter, a set of Status and Error Registers, and six general purpose accumulators (AC0-AC5).

Each accumulator is interpreted to be 32 or 64 bits long depending on the instruction and the status of the Floating Point Processor. For 32-bit instructions only the left-most 32 bits are used, while the remaining 32 bits remain unaffected.

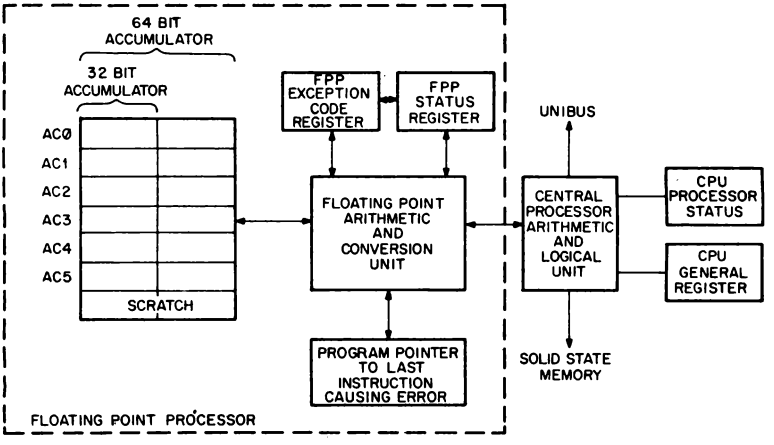


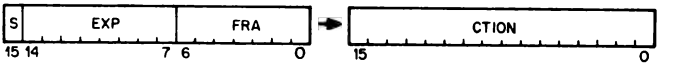
Figure 7-1 Floating Point Processor

The six Floating Point Accumulators are used in numeric calculations and inter-accumulator data transfers; the first four (AC0-AC3) are also used for all data transfers between the FPP and the General Registers or Memory.

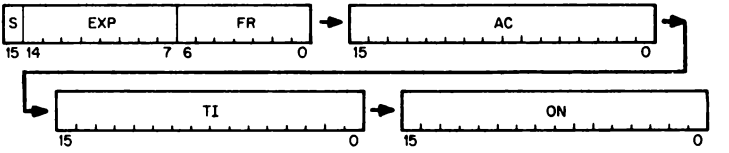
7.4 FLOATING POINT DATA FORMATS

The FPP handles two types of floating point data: Single Precision or Floating Mode (F) which is 32 bits long, and Double Precision (D) which is 64 bits long. The exponent is stored in excess 128 (200_8) notation. Exponents from -128 to +127 are therefore represented by the binary equivalent of 0 to 255 ($0\text{-}377_8$). Fractions are represented in sign-magnitude notation with the binary radix point to the left. Numbers are assumed to be normalized and, therefore, the most significant bit is not stored because it is redundant. It is always a 1 except where the exponent is zero, then the complete number is declared to be 0.

F Formats:



D Formats:



S = Sign of Fraction

EXP = Exponent in excess 200_8 notation

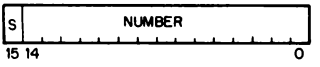
FRACTION = 23 bits in F Format, 55 bits in D Format, + one hidden bit (normalization). Binary Radix point to the left.

The results of a Floating Point operation may be either truncated or rounded off. "Rounding" rounds away from zero and thus increases the absolute value of the number.

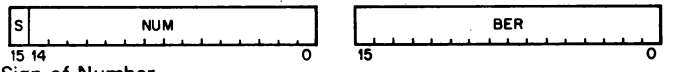
The FPP provides for conversion of Floating Point to Integer Format and vice-versa. The processor thus recognizes single precision integer (I) and double precision integer long (L) numbers.

The numbers are stored in standard two's complement form.

I Format:



L Format:



S = Sign of Number

NUMBER = 15 bits in I Format, 31 bits in L Format.

7.5 FLOATING POINT UNIT STATUS REGISTER

This register provides mode control for the floating point unit, as well as the condition code and error recovery information from the execution of the previous instruction.

Four bits control the modes of operation:

Single/Double - Floating Point numbers can be either single or double precision

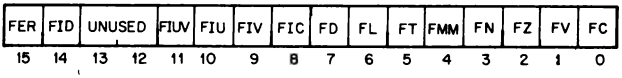
Long/Short - Integer numbers can be 16 bits or 32 bits long.

Truncate/Round - The result of Floating Point operations can be either truncated or rounded off.

Normal/Maintenance - a special maintenance mode is available.

There are four condition codes:

carry, overflow, zero, and negative, which are equivalent to the CPU condition codes, and five error interrupts which can be disabled individually or as a group.



BIT	NAME	DESCRIPTION
15	Floating Error (FER)	Floating Point Error flag. The result of the last operation resulted in a Floating Point Exception and the individual interrupt (FIUV, FIU, FIV, FIC) was enabled.
14	Interrupt Disable (FID)	All FPP interrupts disabled when this bit is set.
13	Not Used	
12	Not Used	
11	Interrupt on Undefined Variable (FIUV)	When set and a -0 is obtained from memory, an interrupt will occur. When clear, -0 can be loaded and used in any arithmetic operation.
10	Interrupt on Underflow (FIU)	When set, Floating Underflow will cause an interrupt. The result of the operation, causing the interrupt, will be correct except for the exponent which will be off by $+400_8$. If the bit is reset and the underflow occurs, the result will be set to zero.

9	Interrupt on Overflow (FIV)	When set, Floating Overflows will cause an interrupt. The result of the operation causing the interrupt will be correct except for the exponent which will be off by $+400_8$. If the bit is reset, the result of the operation will be the same as detailed above but no interrupt will occur.
8	Interrupt on Integer Conversion Error (FIC)	When set, and the STCFI (Store and Convert Floating to Integer) instruction causes FC to be set, an interrupt will occur. If the interrupt occurs, the destination is set to 0 and all other registers are left untouched. If the bit is reset, the result of the operation will be the same as detailed above, but no interrupt will occur.
7	Floating Double Precision Mode (FD)	Determines the precision that is used for Floating Point calculations. When set, Double precision is assumed; when reset Floating precision is used.
6	Floating Long Integer Mode (FL)	Active in conversion between Integer and Floating Point format. When set, the Integer format assumed is Double Precision two's complement (i.e. 31 bits + sign). When reset, the integer format is assumed to be Single Precision two's complement (i.e. 15 bits + sign).
5	Floating Truncate Mode (FT)	When set, causes the result of any arithmetic operation to be truncated when reset the results are rounded.
4	Floating Maintenance Mode (FMM)	
3	Floating Negative (FN)	The result of the last operation was negative.
2	Floating Zero (FZ)	The result of the last operation was zero.
1	Floating Overflow (FV)	The result of the last operation resulted in an arithmetic overflow.
0	Floating Carry (FC)	The result of the last operation resulted in a carry of the most significant bit. This can only occur in integer-Floating conversions.

7.6 FEC REGISTER: ERROR DETECTION

One Interrupt vector is assigned to take care of all floating point exceptions (location 244). The eight possible errors causing the trap are coded in a four bit register, the FPP's Exception Code, "FEC", Register.

The error assignments are as follows:

0	Not used in PDP-11/45
2	Floating OP Code Error
4	Floating Divide by Zero
6	Floating Integer Conversion Error
8	Floating Overflow
10	Floating Underflow
12	Floating Undefined Variable
14	Maintenance Trap

7.7 FLOATING POINT PROCESSOR INSTRUCTION ADDRESSING

Floating Point Processor instructions use the same type of addressing as the Central Processor instructions. A source or destination operand is specified by designating one of eight addressing modes and one of eight central processor general registers to be used in the specified mode. The modes of addressing are the same as those of the central processor except for mode 0. In mode 0 the operand is located in the designated Floating Point Processor Accumulator, rather than in a Central processor general register. The modes of addressing:

0 = Direct Floating Accumulator

1 = Deferred

2 = Auto-increment

3 = Auto-increment deferred

4 = Auto-decrement

5 = Auto-decrement deferred

6 = Indexed

7 = Indexed deferred

Autoincrement and autodecrement operate on increments and decrements of 4 for F Format and 10_8 for D Format.

In mode 0, the user can make use of all six FPP accumulators (AC0 - AC5) as his source or destination. In all other modes, which involve transfer of data from memory or the general register, the user is restricted to the first four FPP accumulators (AC0 -AC3).

In immediate addressing (Mode 2, R7) only 16 bits are loaded or stored.

7.8 FLOATING POINT PROCESSOR INSTRUCTION TIMING

The following table represents execution times of the Floating Point Processor for AC - AC operations (Address Mode 0).

Instruction	Execution Time in Microseconds			
	Single Precision		Double Precision	
	min	max	min	max
ADDX	3.2	4.6	3.8	6.2
SUBX	3.2	4.6	3.8	6.2
MULX	4.6	6.6	6.6	12.0
DIVX	4.6	10.0	6.6	18.4

Minimum and maximum times for FPP instruction execution are given in this chapter using the definition Mode 0-AC operations operating in bipolar memory.

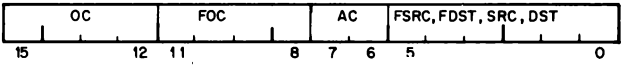
For more detailed information on FPP instruction timing and FPP/CPU interaction consult Appendix B

7.9 FLOATING POINT INSTRUCTIONS

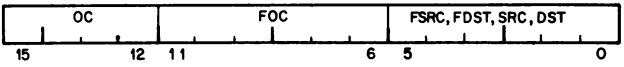
Each instruction that references a floating point number can operate on floating or double precision numbers depending on the state of the FD mode bit. In a similar fashion, there is a mode bit FL that determines whether a 32-bit integer (FL = 1) or a 16-bit integer (FL = 0) is used in conversion between integer and floating point representation. FSRC and FDST use floating point addressing modes, SRC and DST use CPU addressing Modes.

Floating Point Instruction Format

Double Operand Addressing



Single Operand Addressing



OC = Op Code = 17

FOC = Floating Op Code

AC = Accumulator

FSRC, FDST use FPP Address Modes

SRC, DST use CPU Address Modes

General Definitions:

XL = largest fraction that can be represented:

$1-2^{-24}$; FD = 0

$1-2^{-56}$; FD = 1

XLL = smallest number that is not identically zero 2^{-128}

XUL = largest number that can be represented: 2^{127}*XL

JL = largest integer that can be represented:

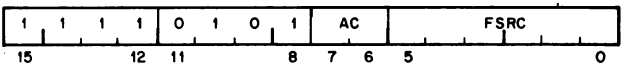
$2^{15}-1$ If FL = 0 $2^{31}-1$ If FL = 1

LDF
LDD

750 ns
750 ns

Load Floating/Double

172(AC + 4)FSRC



Operation: AC← (FSRC)

Condition Codes:
FC←0
FV←0
FZ←1 if (AC) = 0 else FZ←0
FN←1 if (AC)<0 else FN←0

Description: Load Single or Double Precision Number into Accumulator

ISP:

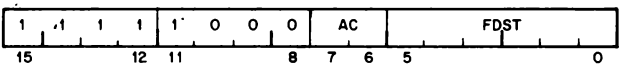
```
LDF:  LDD:
      Fac ← FPS'; .ext                    move source to floating accumulator
     (Fac<0) ⇒    . ← 1 else FN ← 0);     negative?
     (Fac=0) ⇒ (FZ ← 1 else FZ ← 0);     zero?
      FV ← 0;                             clear overflow
      FC ← 0                              clear carry
```

750 ns
750 ns

STF
STD

Store Floating/Double

174ACFDST



Operation: FDST←(AC)

Condition Codes: FC←FC
FV←FV
FZ←FZ
FN←FN

Description: Store Single or Double Precision Number from Accumulator

ISP:

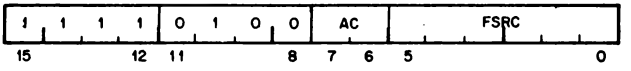
```
STF:  STD:
      FPD' ← Fac                    move floating accumulator to
                                     destination
```

ADDF
ADDD

3.15 μs · 4.55 μs
3.75 μs · 6.1 μs

Add Floating/Double

172ACFSRC



Operation: AC←(AC) + (FSRC) If [(AC) + (FSRC)] >XLL or FIU = 1, else AC←0

Condition Codes:
FC←0
FV←1 If (AC)≥XUL else FV←0
FZ←1 If (AC)=0 else FZ←0
FN←1 If (AC)<0 else FN←0

Description: Add the contents of FSRC to the contents of accumulator. In Single or Double Precision result is in accumulator unless Underflow occurs and the interrupt is not enabled; in this case AC is set to 0.

ISP:

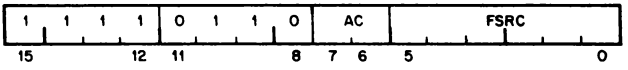
```
ADDF:  ADDD:
      Fr ← Fac + FPS'; next                                  move sum of accumulator and source
                                                               to temporary register
    ((Fr>XLL) ∨ (FIU=1)) ⇒ (Fac ← Fr else Fac ← 0); next     store result if no underflow, zero
                                                               otherwise
     (Fac<0) ⇒ (FN ← 1 else FN ← 0);                        negative?
     (Fac=0) ⇒ (FZ ← 1 else FZ ← 0);                        zero?
     (Fac>XUL) ⇒ (FV ← 1 else FV ← 0);                      overflow?
      FC ← 0                                                clear carry
```

3.15 μs - 4.55 μs
3.75 μs - 6.1 μs

SUBF
SUBD

Subtract Floating/Double

173ACFSRC



Operation: AC←(AC)-(FSRC) If [(AC)-(FSRC)]-XLL or FIU = 1 else AC←0

Condition Codes:
FC←0
FV←1 If (AC)>XUL else FV←0
FZ←1 If (AC) = 0 else FZ←0
FN←1 If (AC)<0 else FN←0

Description: Subtract the contents of FSRC from the accumulator in Single or Double Precision. Result is in accumulator unless Floating Underflow occurs and the interrupt is not enabled, in this case AC is set to 0.

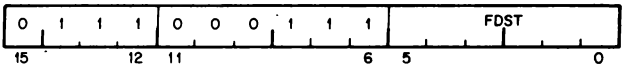
ISP:

```
SUBF:  SUBD:
      Fr ← Fac - FPS'; next                                  difference of Fac and source to Fr
    ((Fr<XLL) ∨ (FIU=1)) ⇒ (Fac ← Fr else Fac ← 0); next     store result if no underflow, zero
                                                             otherwise
     (Fac<0) ⇒ (FN ← 1 else FN ← 0);                         negative?
     (Fac=0) ⇒ (FZ ← 1 else FZ ← 0);                         zero?
     (Fac>XUL) ⇒ (FV ← 1 else FV ← 0);                       overflow?
      FC ← 0                                                 clear carry
```

NEGF
NEGD

1.75 μs
1.75 μs

Negate Floating/Double 1707FDST



Operation: FDST←-(FDST)

Condition Codes: FC←0
FV←0
FZ←1 If (FDST) = 0 else FZ←0
FN←1 If (FDST)<0 else FN←0

Description: Negate Floating or Double Precision number, store result in same location. (FDST)

ISP:

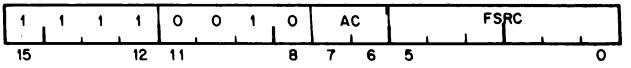
```
NEGF:  NEGD:
     Fr ← - FPD; next                    negative of destination to Fr
     FPD' ← Fr;                          store result
    (Fr<0) ⇒ (FN ← 1 else FN ← 0);       negative?
    (Fr=0) ⇒ (FZ ← 1 else FZ ← 0);       zero?
     FV ← 0;                             clear overflow
     FC ← 0                              clear carry
```

4.55 μs - 6.55 μs
6.55 μs - 11.95 μs

MULF
MULD

Multiply Floating/Double

171ACFSRC



Operation: AC←(AC)*(FSRC) If [(AC)*(FSRC)]>XLL or FIU = 1, else AC←0

Condition Codes:
FC←0
FV←1 If AC>XUL else FV←0
FZ←1 If (AC) = 0 else FZ←0
FN←1 If (AC)<0 else FN←0

Description: Multiply the contents of the selected accumulator by the contents of FSRC. Store result in accumulator unless Floating Underflow occurs without the interrupt enabled in this case AC is set to 0.

ISP:

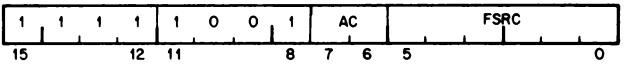
```
MULF:  MULD:
      Fr ← Fac * FPS'; next                                  product of Fac and source to Fr
    ((Fr>XLL) ∨ (FIU=1)) ⇒ (Fac ← Fr else Fac ← 0); next     store result if no underflow, zero
                                                             otherwise
     (Fac<0) ⇒ (FN ← 1 else FN ← 0);                        negative?
     (Fac=0) ⇒ (FZ ← 1 else FZ ← 0);                        zero?
     (Fac>XUL) ⇒ (FZ ← 1 else FV ← 0);                      overflow?
      FC ← 0                                                 clear carry .
```

DIVF
DIVD

4.55 μs · 9.95 μs
6.55 μs · 18.35 μs

Divide Floating / Double

174(AC + 4)FSRC



Operation: If (FSRC) ≠ 0
AC←(AC)/(FSRC) If [(AC)/(FSRC)]>XLL or FIU = 1,
else AC←0
If (FSRC) = 0 registers, including AC, untouched

Condition Codes: FC←0
FV←1 If (AC) >XUL else FV←0
FZ←1 If (AC) = 0 else FZ←0
FN←1 If (AC) <0 else FN←0

Description: If the contents of FSRC are not equal to zero, divide the accumulator by (FSRC) and store the result in the accumulator unless Floating Underflow occurs and the interrupt is not enabled in this case the AC is set to 0. If attempt is made to divide by zero, accumulator is left unchanged and FEC Register is set to 4.

ISP:

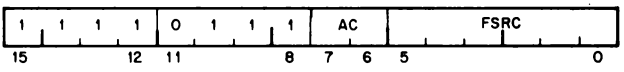
```
DIVF:  DIVD:
     (FPS=0) ⇒ (FEC ← 4);                                    set error 4 if divisor is zero
     (FPS≠0) ⇒ (                                             if non-zero divisor:
        Fr ← Fac/FPS'; next                                  result of division to Fr
        ((Fr>XLL) ∨ (FIU=1)) ⇒ (Fac ← Fr else Fac ← 0); next store result if no underflow, zero
                                                               otherwise

        (Fac<0) ⇒ (FN ← 1 else FN ← 0);                      negative?
        (Fac=0) ⇒ (FZ ← 1 else FZ ← 0);                      zero?
        (Fac>XUL) ⇒ (FV ← 1 else FV ← 0);                    overflow?
        FC ← 0)                                              clear carry
```

1.95 μs - 2.55 μs
2.35 μs - 2.95 μs

CMPF
CMPD

Compare Floating/Double

173(AC + 4)FSRC



Operation: (FSRC) - (AC)

Condition Codes:
FC←0
FV←0
FZ←1 If (FSRC)-(AC) = 0 else FZ←0
FN←1 If (FSRC)-(AC)<0 else FN←0

Description: Compare the contents of FSRC with the accumulator. Set the appropriate floating point condition codes; FSRC and the accumulator are left unchanged.

ISP:

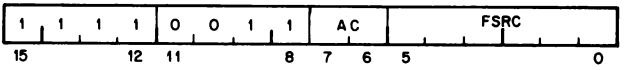
```
CMPF:  CMPD:
    Fr ← FPS' - Fac; next                    compare affects only condition codes
    (Fr<0) ⇒ (FN ← 1 else FN ← 0);           negative?
    (Fr=0) ⇒ (FZ ← 1 else FZ ← 0);           zero?
    FV ← 0;                                  clear overflow
    FC ← 0                                   clear carry
```

MODF
MODD

6.15 μs - 12.35 μs
8.35 μs - 22.55 μs

Multiply and Integerize Floating/Double

171(AC + 4)FSRC



Operation: AC ν 1←Int[(AC)*(FSRC)] If [(AC)*(FSRC)]> XLL
or FIU = 1, else AC ν 1←0
AC←(AC)*(FSRC)-(AC ν 1) If [(AC)*(FSRC)]> XLL
or FIU = 1, else AC←0

Condition Codes: FC←0
FV←1 If (AC) >XUL else FV←0
FZ←1 If (AC) = 0 else FZ←0
FN←1 If (AC)<0 else FN←0

Description: The product of (AC) and (FSRC) is produced to 48 bits in Floating Mode and 59 bits in Double Mode. The integer part Int[(AC)*(FSRC)] of the product is then found and stored in AC ν 1. AC ν 1 is the FPP Accumulator OR'd with 1. The fractional part is then obtained and stored in AC. Thus if even-numbered Accumulators (0 or 2) are used this instruction uses two accumulators (0 and 1; 2 and 3); whereas if odd-numbered accumulators are used only one Accumulator is used (1:3) and all that is left is the fractional part of the operation. If underflow occurs and the interrupt is not enabled, AC and AC ν 1 are loaded with zero.
NOTE: Multiplication by 10 can be done with zero error allowing decimal digits to be "stripped off" with no loss in precision.

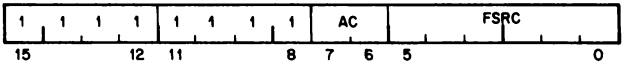
ISP:

```
MODF:  MODD:
     Fr ← Fac * FPS'; next                    product to Fr
     ((Fr>XLL) ∨ (FIU=1)) ⇒                   if no underflow:
        (FAC(ACV1) ← Int[Fr] else              store Int[Fr] in AC ∨ 1
         FAC(ACV1) ← 0); next                  otherwise store zero in AC ∨ 1
     ((Fr>XLL) ∨ (FIU=1)) ⇒                   if no underflow:
        (Fac ← Fr - FAC(ACV1) else             store Fr - Int[Fr]
         Fac ← 0); next                        otherwise store zero
     (Fac<0) ⇒ (FN ← 1 else FN ← 0);          negative?
     (Fac=0) ⇒ (FZ ← 1 else FZ ← 0);          zero?
     (Fac>XUL) ⇒ (FV ← 1 else FV ← 0);        overflow?
     FC ← 0                                   clear carry
```

2.75 μs - 3.15 μs
2.75 μs

LDCDF
LDCFD

Load and convert from Double to Floating or from Floating to Double 177(AC + 4)FSRC



Operation: AC←C_{xy}(FSRC) If [(FSRC)]>XLL or FIU = 1, else AC←0 Where C_{xy} specifies conversion from floating mode x to floating mode y, and x = F and y = D If FD = 0, or x = D and y = F If FD = 1.

Condition Codes: FC←0
FV←1 If (AC)>XUL else FV←0
FZ←1 If (AC) = 0 else FZ←0
FN←1 If (AC)<0 else FN←0

Description: If the current mode is Floating Mode (FD = 0) the source is assumed to be a double-precision number and is converted to single precision. If the Floating Truncate bit is set the number is truncated, otherwise the number is rounded. If the current mode is Double mode (FD = 1) the source is assumed to be a single-precision number and is loaded left justified in the AC. The lower half of the AC is cleared.

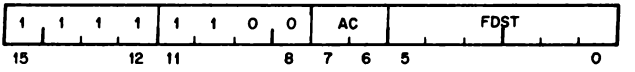
ISP:

```
LDCDF:  LDCFD:
     Fr ← FPS'; next                          move source to Fr
     ((Fr>XLL) V (FIU=1)) ⇒ (                 if no underflow:
        (FD=0) ⇒ (Fac ← CFD(Fr));              convert single to double if FD=0
        (FD=1) ⇒ (Fac ← CDF(Fr));              convert double to single if FD=1
        else Fac ← 0); next                    otherwise store zero result
     (Fac<0) ⇒ (FN ← 1 else FN ← 0);          negative?
     (Fac=0) ⇒ (FZ ← 1 else FZ ← 0);          zero?
     (Fac>XUL) ⇒ (FV ← 1 else FV ← 0);        overflow?
     FC ← 0                                   clear carry
```

STCFD
STCDF

1.35 μs
2.55 μs

Store and convert from Floating to Double or from Double to Floating 176ACFDST



Operation: FDST←C_{xy}(AC) where C_{xy} specifies conversion from floating mode x to floating mode y and x = F and y = D If FD = 0, or x = D and y = F If FD = 1

Condition Codes:
FC←0
FV←1 If (AC)>XUL else FV←0
FZ←1 If (AC) = 0 else FZ←0
FN←1 If (AC)<0 elseFN←0

Description: If the current mode is Floating, the Accumulator is stored left justified in FDST and the lower half is cleared; otherwise in Double Precision, the contents of the accumulator are converted to single precision, truncated or rounded depending on the state of FT and stored in FDST.

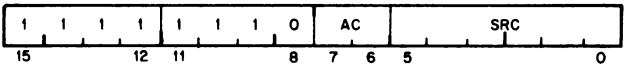
ISP:

```
STCFD:  STCDF:
    (FD=0) ⇒ (FPD' ← CFD(Fac));          convert single to double if FD=0
    (FD=1) ⇒ (FPD' ← CDF(Fac));          convert double to single if FD=1
    (Fac<0) ⇒ (FN ← 1 else FN ← 0);      negative?
    (Fac=0) ⇒ (FZ ← 1 else FZ ← 0);      zero?
    (Fac>XUL) ⇒ (FV ← 1 else FV ← 0);    overflow?
    FC ← 0                               clear carry
```

4.15 μs - 5.15 μs
4.35 μs - 5.3 μs
4.15 μs - 5.95 μs
4.35 μs - 6.15 μs

LDCIF
LDCID
LDCLF
LDCLD

Load and Convert Integer or Long Integer to Floating or Double Precision 177ACSRC



Operation: AC←C_{jx}(SRC) where C_{jx} specifies conversion from integer mode j to floating mode x and j = I if FL = 0 or L if FL = 1 and x = F if FD = 0, or D if FD = 1.

Condition Codes:
FC←0
FV←0
FZ←1 If (AC) = 0 else FZ←0
FN←1 If (AC)<0 else FN←0

Description: Conversion is performed on the contents of SRC from a 2's compliment Integer with precision j to a floating point number of precision x. Note that j and x are determined by the state of the mode bits FL and FD: i.e.J = I or L, and X = F or D.
When a 32 bit Integer is specified (L mode) and (SRC) has an addressing mode of 0, or immediate addressing mode is specified, the 16 bits of the source register are left justified and the remaining 16 bits loaded with zeros before conversion. In the case of LDCLF the fraction is truncated and only the highest 24 significant bits are used.

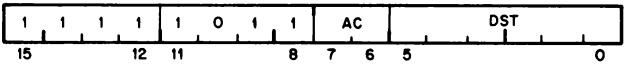
ISP:

```
LDCIF:  LDCID:  LDCLF:  LDCLD:
     ((FL=0) ⇒ (Fr ← FS');                          get 16 bit source if FL=0
      (FL=1) ⇒ (                                    if FL=1:
         (dm=0) ⇒ (FR ← FS');                        get 16 bit source if mode zero
         (dm≠0) ⇒ (Fr ← FS'□nw')));  next            get 32 bit source otherwise
    (((FD=0) ∧ (FL=0)) ⇒ (Fac ← CIF(Fr));           convert integer to floating
     ((FD=0) ∧ (FL=1)) ⇒ (Fac ← CID(Fr));           convert integer to floating double
     ((FD=1) ∧ (FL=0)) ⇒ (Fac ← CLF(Fr));           convert long integer to floating
     ((FD=1) ∧ (FL=1)) ⇒ (Fac ← CLD(Fr)));  next    convert long integer to floating double
     (Fac<0) ⇒ (FN ← 1 else FN ← 0);                negative?
     (Fac=0) ⇒ (FZ ← 1 else FZ ← 0);                zero?
     FV ← 0;                                        clear overflow
     FC ← 0                                         clear carry
```

STCFI
STCFL
STCDI
STCDL

4.75 μs - 5.75 μs
5.55 μs - 6.55 μs
5.55 μs - 6.55 μs
5.55 μs - 6.55 μs

Store and Convert from Floating or Double to integer or Long Integer 175(AC + 4)DST



Operation: DST←C_{xj}(AC) If -JL-1<C_{xj}(AC)<JL, else DST←0 where C_{xj} specifies conversion from floating mode x to integer mode j and j = I if FL = 0 or L if FL = 1 and x = F if FD = 0, or D if FD = 1

Condition Codes: C← FC←0 If -JL-1<C_{xj}(AC)<JL else FC←1
V← FV←0
Z← FZ←1 If (DST) = 0 else FZ←0
N← FN←1 If (DST)<0 else FN←0

Description: Conversion is performed from a floating point representation of the data in the accumulator to an integer representation. When the conversion is to a 32 bit word (L mode) and an address mode of 0, or immediate addressing mode, is specified, only the most significant 16 bits are stored in the destination register. If the operation is out of the integer range selected by FL, FC is set to 1 and the contents of the DST are set to 0.

ISP:

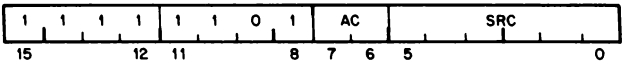
```
STCFI:  STCFL:  STCDI:  STCDL:
    (((FD=0) ∧ (FL=0)) ⇒ (Fr ← CFI(Fac));          convert floating to integer
     ((FD=0) ∧ (FL=1)) ⇒ (Fr ← CFL(Fac));          convert floating to long integer
     ((FD=1) ∧ (FL=0)) ⇒ (Fr ← CDI(Fac));          convert floating double to integer
     ((FD=1) ∧ (FL=1)) ⇒ (Fr ← CDL(Fac))); next    convert floating double to long integer
     ((Fr<JL) ∧ (Fr>-JL-1)) ⇒                      if no carry:
        (FC ← 0 else (FC ← 1; Fr ← 0)); next        clear carry, otherwise set carry-result zero
     (FL=0) ⇒ (FD' ← Fr);                          16 bit result if FL=0
     (FL=1) ⇒ (                                    if FL=1:
        (dm=0) ⇒ (FD' ← Fr<15:0>);                  16 bit result if mode zero
        (dm≠0) ⇒ (FD'□nw' ← Fr<31:0>));             32 bit result otherwise
     (Fr<0) ⇒ (FN ← 1 else FN ← 0);                negative?
     (Fr=0) ⇒ (FZ ← 1 else FZ ← 0);                zero?
     FV ← 0                                        clear overflow
```

2.15 μs

LDEXP

Load Exponent

176(AC + 4)SRC



Operation: AC SIGN←(AC SIGN)
AC EXP←(SRC) + 200

Condition Codes: FC←0
FV←0
FZ←1 If (AC) = 0 else FZ←0
FN←1 If (AC) < 0 else FN←0

Description: Load Exponent from SRC into Accumulator. Convert (SRC) from 2's complement to excess 200_8 notation.

ISP:

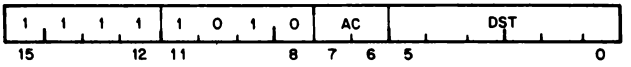
```
LDEXP:
    Fac<62:55> ← FS' + 200₈; next          load exponent from source, excess 200₈
    (Fac<0) ⇒ (FN ← 1 else FN ← 0);        negative?
    (Fac=0) ⇒ (FZ ← 1 else FZ ← 0);        zero?
    FV ← 0;                                clear overflow
    FC ← 0                                 clear carry
```

STEXP

2.75 μs

Store Exponent

175ACDST



Operation: DST←AC EXPONENT - 200

Condition Codes:
C← FC←0
V← FV←0
Z← FZ←1 if (DST) = 0 else FZ←0
N← FN←1 if (DST)<0 else FN←0

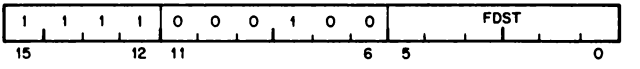
Description: Store accumulator's exponent in DST, convert it from excess 200_8 notation to 2's complement.

```
STEXP:
    Fr ← Fac:62:55  - 200₈; next        convert exponent to 2's complement
    FD' ← Fr; next                      store result
    (Fr<0) ⇒ (FN ← 1 else FN ← 0);      negative?
    (Fr=0) ⇒ (FZ ← 1 else FZ ← 0);      zero?
    FV ← 0;                             clear overflow
    FC ← 0                              clear carry
```

1.35 μs
1.75 μs

CLRF
CLRD

Clear Floating/Double 1704FDST



Operation: FDST←0

Condition Codes: FC←0
FV←0
FZ←1
FN←0

Description: Set FDST to 0. Set FZ condition code.

ISP:

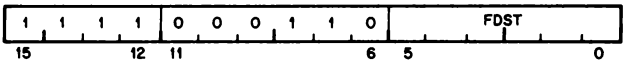
```
CLRF:  CLRD:
    FPD' ← 0;                    clear destination
    FN ← 0;                      clear negative
    FZ ← 1;                      set zero
    FV ← 0;                      clear overflow
    FC ← 0                       clear carry
```

ABSF
ABSD

1.35 μs
1.35 μs

Make Absolute Floating / Double

1706FDST



Operation: FDST←-(FDST) If (FDST)<0 else FDST←(FDST)

Condition Codes:
FC←0
FV←0
FZ←1 If (FDST) = 0 else FZ←0
FN←0

Description: Set the contents of FDST to its absolute value.

ISP:

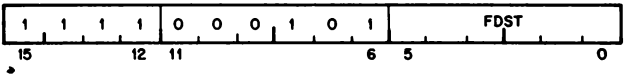
```
ABSF:  ABSD:
    Fr ← FPD; next                          move destination to Fr
   (Fr<0) ⇒ (Fr ← - Fr); next               negate if negative
    FPD' ← Fr;                              store result
    FN ← 0;                                 clear negative
   (Fr=0) ⇒ (FZ ← 1 else FZ ← 0);           zero?
    FV ← 0;                                 clear overflow
    FC ← 0                                  clear carry
```

1.5 μs
1.5 μs

TSTF
TSTD

Test Floating/Double

1705FDST



Operation: FDST←(FDST)

Condition Codes:
FC←0
FV←0
FZ←1 If (FDST) = 0 else FZ←0
FN←1 If (FDST)<0 else FN←0

Description: Set the Floating Point Processor's Condition Codes according to the contents of FDST.

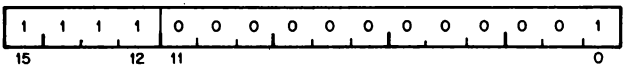
ISP:

```
TSTF:  TSTD:
    Fr ← FPD'; next                          move destination to Fr
   (Fr<0) ⇒ (FN ← 1 else FN ← 0);            negative?
   (Fr=0) ⇒ (FZ ← 1 else FZ ← 0);            zero?
    FV ← 0;                                  clear overflow
    FC ← 0                                   clear carry
```

SETF

.75 μs

Set Floating Mode 170001



Operation: FD←0

Description: Set the FPP in Single Precision Mode

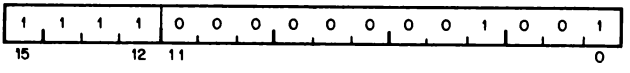
ISP:

```
SETF:
    FD ← 0                                  clear floating double
```

SETD

.75 μs

Set Floating Double Mode 170011



Operation: FD←1

Description: Set the FPP in Double Precision Mode

ISP:

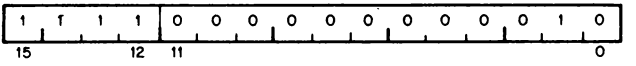
```
SETD:
    FD ← 1                                  set floating double
```

.75 μs

SETI

Set Integer Mode

170002



Operation: FL←0

Description: Set the FPP for Integer Data

ISP:

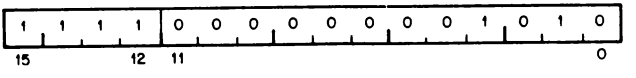
```
SETI:
    FL ← 0                                  clear long integer
```

.75 μs

SETL

Set Long Integer Mode

170012



Operation: FL←1

Description: Set the FPP for Long Integer Data

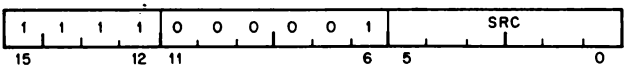
ISP:

```
SETL:
    FL ← 1                                  set long integer
```

LDFPS

1.35 μs

Load FPPs Program Status 1701SRC



Operation: FPS←(SRC)

Description: Load FPP's Status from SRC.

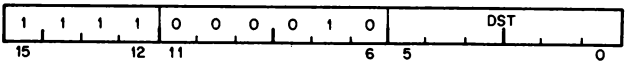
ISP:

```
LDFPS:
    FPSR ← FS'                                  load FPP status register
```

STFPS

1.35 μs

Store FPPs Program Status 1702DST



Operation: DST←(FPS)

Description: Store FPP's Status in DST

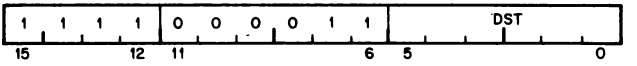
ISP:

```
STFPS:
    FD' ← FPSR                                  store FPP status register
```

1.35 μs

STST

Store FPPs Status 1703DST



Operation: DST←(FEC)
DST + 2←(FEA)

Description: Store the FEC and then the FPP's Exception Address Pointer in DST and DST + 2

Note: If destination mode specifies a general register or immediate addressing, only the FEC is saved.

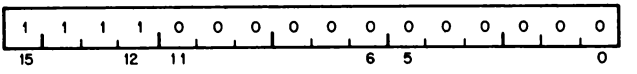
ISP:

```
STST:                                   store floating error register
    FD' ← FEC                           store floating PC
    nw' ← FPC
```

7.75 μs

CFCC

Copy Floating Condition Codes 170000



Operation: C←FC
V←FV
Z←FZ
N←FN

Description: Copy FPP Condition Codes into the CPU's Condition Codes.

ISP:

```
CFCC:
    N ← FN;                             move FN to N
    Z ← FZ;                             move FZ to Z
    V ← FV;                             move FV to V
    C ← FC                              move FC to C
```

Figure 8-1 System Operator's Console

CHAPTER 8

THE SYSTEM OPERATOR'S CONSOLE

The PDP-11/45 System Operator's Console is designed for convenient system control. A complete set of function switches and display indicators provide comprehensive status monitoring and control facilities.

The System Operator's Console is illustrated in Figure 8.1.

8.1 CONSOLE ELEMENTS

The PDP-11/45 System Operator's Console provides the following facilities:

1) A System Key Switch (OFF/ON/LOCK)
2) A bank of 7 indicator lights, indicating the following Central Processor states: RUN, PAUSE, MASTER(UNIBUS), USER, SUPERVISOR, KERNEL, DATA.
3) An 18-bit Address Register display
4) An Addressing Error indicator light (ADRS ERR)
5) A 16-bit Data Register display
6) An 18-bit Switch Register
7) Control knobs
    a) Address Display Select
        1. USER I VIRTUAL
        2. USER D VIRTUAL
        3. SUPERVISOR I VIRTUAL
        4. SUPERVISOR D VIRTUAL
        5. KERNEL I VIRTUAL
        6. KERNEL D VIRTUAL
        7. PROGRAM PHYSICAL
        8. CONSOLE PHYSICAL

b) Data Display Select

1. DATA PATHS
2. BUS REGISTER
3. FPP μADRS.CPU μADRS.
4. DISPLAY

8) Control Switches

a) LOAD ADRS (Load Address)

b) EXAM (Examine)

c) REG EXAM (Register Examine)

d) CONT (Continue)

e) ENABLE/HALT

f) S-INST/S-BUS CYCLE (Single Instruction/Single Bus Cycle)

g) START

h) DEPOSIT

i) REG DEPOSIT (Register Deposit)

8.2 SYSTEM POWER SWITCH

The System Power Switch controls Central Processor power as follows:

OFF	Power off for CPU Solid-State Memory still receives power in order to insure data retention.
POWER	Power ON for CPU - normal use all console controls operable.
PANEL LOCK	Power ON for CPU. All console controls not operable except switch register.

Note: Since the theory of operation of high speed solid state memory involves the retention of a capacitive charge, it is essential that power be continually supplied in order to insure full data retention during those periods when the CPU power is OFF. When this facility is not required, Memory Power may be discontinued by flipping the Master Power switch in the rear of the CPU mounting cabinet to OFF.

8.3 CENTRAL PROCESSOR STATE INDICATORS

This bank of indicator lights shows the current major system state as follows:

RUN	The CPU is executing program instructions. If the instruction being executed is a WAIT instruction, the RUN light will be on. The CPU will proceed from the WAIT on receipt of an external interrupt, or on console intervention.
PAUSE	The CPU is inactive because: 1) The current instruction execution has been completed as far as possible without more data from the UNIBUS and the CPU is waiting to regain control of the UNIBUS (UNIBUS mastership) (see MASTER state.) OR 2) The CPU has been HALTed from the System Operator's Console.
MASTER	The CPU is in control of the UNIBUS (UNIBUS Master). The CPU relinquishes control of the UNIBUS during DMA and NPR data transfers.
USER	The CPU is executing program instructions in USER mode. When the Memory Segmentation Unit is enabled all address references are in USER Virtual Address Space.
SUPERVISOR	The CPU is executing program instructions in SUPERVISOR mode. When the Memory Segmentation Unit is enabled, all address references are in SUPERVISOR Virtual Addressing space.
KERNEL	The CPU is executing program instructions in KERNEL mode. When the Memory Segmentation Unit is enabled, all address references are in KERNEL Virtual Addressing space.
DATA	If on a 1 the last memory reference was to D address space in the current CPU mode. If a 0, the last memory reference was to I address space in the current CPU mode.

8.4 ADDRESS DISPLAY REGISTER

The Address Display Register is primarily a software development and maintenance aid. The contents of this 18-bit indicator are controlled by the Address Display Select knob as follows:

VIRTUAL	The Address Display Register indicates the current address reference as a 16-bit Virtual Address when the Memory Segmentation Unit is enabled, otherwise it indicates the true 16-bit Physical Address. Bits 17 and 16 will be off unless the Memory Segmentation Unit is disabled AND the current address references some UNIBUS device register in the uppermost 4K of basic address space (i.e. 28K-32K).
PROGRAM PHYSICAL	The Address Display Register indicates the current address reference as a true 18-bit Physical Address.
CONSOLE PHYSICAL	The Address Display Register indicates the current address reference as a 16-bit Virtual Address when the Memory Segmentation Unit is enabled otherwise it indicates the true 16-bit Physical Address. Bits 17 and 16 indicate the contents of corresponding bits of the Switch Register as of the last LOAD ADRS console operation.

8.5 ADDRESSING ERROR DISPLAY

This 1-bit display indicates the occurrence of any addressing errors. The following address references are invalid:

1. Non-existent memory
2. Access Control violations
3. Unassigned memory segments

(See chapter 4: Memory Management)

8.6 DATA DISPLAY REGISTER

The Data Display Register is primarily a hardware maintenance facility. The contents of this 16-bit indicator are controlled by the Data Display Select knob as follows:

DATA PATHS	The Data Display Register indicates the current output of the PDP-11/45 Arithmetic/Logical Unit subsystem.
BUS REGISTER	The Data Display Register indicates the current output of the PDP-11/45 CPU (UNIBUS I, II and the EXPRESS BUS).
FPP μADRS.CPU μADRS.	The Data Display Register indicates the current ROM address in the low byte. FPP control micro-program (bits 15-8), or the CPU control micro-program (bits 7-0).
DISPLAY	The Data Display Register indicates the current contents of the 16-bit write-only "Switch Register"located at Physical Address 777750. This register is generally used to display diagnostic information, although it can be used for any meaningful purpose.

8.7 SWITCH REGISTER

The functions of this 18-bit bank of switches are determined by:

1) Control Switches

2) Address Display Select knob

These functions will be described in the next section along with the appropriate control switch.

Note that the current setting of the Switch Register may be read under program control from a read-only register at Physical Address 777750.

8.8 CONTROL SWITCHES

8.8.1 LOAD ADRS (Load Address)

When the LOAD ADRS switch is depressed the contents of the Switch Register are loaded into the CPU Bus Address Register and displayed in the Address Display Register lights. If the Memory Segmentation Unit is disabled the address displayed is the true Physical Address.

If the Memory Segmentation Unit is enabled the interpretation of the address indicated by the Switch Register is determined by the Address Display Select knob.

Note that the LOAD ADRS function does not distinguish between PROGRAM PHYSICAL and CONSOLE PHYSICAL.

8.8.2 EXAM (Examine)

Depressing the EXAM switch causes the contents of the current location specified in the CPU Bus Address Register to be displayed in the DATA Display Register.

Depressing the EXAM switch again causes a EXAM-STEP operation to occur. the result is the same as the EXAM except that the contents of the CPU Bus Address Register are incremented by two before the current location has been selected for display. An EXAM-STEP will not cross a 32K memory block boundary.

An EXAM operation which causes an ADRS ERR (Addressing Error) must be corrected by performing a new LOAD ADRS operation with a valid address.

8.8.3 REG EXAM (Register Examine)

Depressing the REG EXAM switch causes the contents of the General Purpose Register specified by the low order five bits of the bus address register to be displayed in the Data Display Register.

The Switch Register is interpreted as follows:

Contents	Register Displayed
0-5	General Registers 0-5 (set 0)
6	Kernel Mode Register 6
7	Program Counter
10_8 - 15_8	General Register 0-5 (set 1)
16_8	Supervisor Mode Register 6
17_8	User Mode Register R6

8.8.4 CONT (Continue)

Depressing the CONT switch causes the CPU to resume executing instructions or bus cycles at the address specified in the Program Counter (Register 7-(PC)). The CONT switch has no effect when the CPU is in RUN state.

The function of the CONT switch is modified by the setting of the ENABLE/HALT and S/INST-S/BUS cycle switches as follows:

ENABLE (up)	CPU resumes normal operation under program control.
HALT (down)	A. S/INST (up) - CPU executes next instruction then stops. B. S/BUS cycle (down) - CPU executes next address reference then stops.

8.8.5 ENABLE/HALT

The ENABLE/HALT switch is a two-position switch with the following functions:

ENABLE (up)	The CPU is able to perform normal operations under program control.
HALT (down)	The CPU is stopped and is only operable by the console switches.

The setting of the ENABLE/HALT switch modifies the function of the CONTINUE (8.8.4) and START (8.8.7) switches.

8.8.6 S/INST - S/BUS CYCLE (Single Instruction/Single Bus Cycle)

The S/INST-S/BUS CYCLE switch effects only the operation of the CONTINUE switch as described in section 8.8.4. This switch has no effect on any switches when the ENABLE/HALT switch is set to ENABLE.

8.8.7 START

The functions of the START switch depend upon the setting of the ENABLE/HALT switch as follows:

ENABLE	Depressing the START switch causes the CPU to start executing program instructions at the address specified by the current contents of the CPU Bus Address Register. The START switch has no effect when the CPU is in RUN state.
HALT	Depressing the START switch causes a console reset to occur.

8.8.8 DEP (Deposit)

Raising the DEP switch causes the current contents of the Switch Register to be deposited into the address specified by the current contents of the CPU Bus Address Register.

Raising the DEP switch again causes a DEP-STEP operation to occur. The result is the same as the DEP except that the contents of the CPU Bus Address Register are incremented by two before the current location has been selected for the deposit operation. A DEP-STEP will not cross a 32K memory block boundary.

A DEP operation which causes an ADRS ERR (addressing Error) is aborted and must be corrected by performing a new LOAD ADRS operation with a valid address.

8.8.9 REG DEPOSIT (Register Deposit)

Raising the REG DEP causes the contents of the Switch Register to be deposited into the General Purpose Register specified by the current contents of the CPU Bus Address Register.

The CPU Bus Address Register should have been previously loaded by a LOAD ADRS operation according to the Switch register settings described in REG EXAM (8.8.3)

NOTE: The EXAM and DEP switches are coupled to enable an EXAM-DEP-EXAM sequence to be carried out on a location, without having to do a LOAD ADRS. The following sequence is possible:

EXAM

DEP ADDRESS A

EXAM

STEP EXAM

DEP ADDRESS A + 1

EXAM

APPENDIX A

A DESCRIPTION OF THE PDP-11
USING THE INSTRUCTION SET PROCESSOR (ISP) NOTATION[1]

ISP is a language (or notation) which can be used to define the action of a computer's instruction set. It defines a computer, including console and peripherals, as seen by a programmer. It has two goals: to be precise enough to constitute the complete specification for a computer and to still be highly readable by a human user for purposes of reference, such as this manual. The main part of the manual contained an English language description of the PDP-11, using ISP expressions as support in defining each instruction. This appendix contains an ISP description of the PDP-11, using a few English language comments as support.

The following brief introduction to the notation is given using examples from the PDP-11 Model 20 ISP description. The complete PDP-11 description follows the introduction.

A processor is completely defined at the programming level by giving its instruction set and its interpreter in terms of basic operations, data types and the system's memory. For clarity the ISP description is usually given in a fixed order:

- Declare the system's memory:

  Processor state (the information necessary to restart the processor if stopped between instructions, e.g., general registers, PC, index registers)

  Primary memory state (the memory directly addressable from the processor)

  Console state (any external keys, switches, lights, etc., that affect the interpretation process)

  Secondary memory (the disks, drums, dectapes, magnetic tapes, etc.)

  Transducer state (memory available in any peripheral devices that is assumed in the instructions of the processor)

  Declare the instruction format
  Define the operand address calculation process
  Declare the data types
  Declare the operations on the data types
  Define the instruction interpretation process including interrupts, traps, etc.
  Define the instruction set and the instruction execution process (provides an ISP expression for each instruction)

Thus, the computer system is described by first declaring memory, data-types and primitive data operations. The instruction interpreter and the instruction-set is then defined in terms of these entities.

The ISP notation is similar to that used in higher level programming languages. Its statements define entities by means of expressions involving other entities in the system. For example, an instruction to increment (add-one) to memory would be

```
Increment := (M[x] ← M[x] + 1);          add one to memory, x
```

This defines an operation, called "increment", that takes the contents of memory M at an address, x, and replaces it with a value one higher. The := symbol simply assigns a name (on the left) to stand for the expression (on the right). English language comments are given in italics. Table 1 gives a reference list of notations, which are illustrated below.

ISP expressions are inherently interpreted in parallel, reflecting the underlying parallel nature of hardware operations. This is an important difference between ISP and standard programming languages, which are inherently serial. For example, in

[1] The notation derived and used in the book, Computer Structures: Readings and Examples, McGraw-Hill, 1971 by C. Gordon Bell and Allen Newell. The book contains ISP's of 14 computers.

```
Z := (M[x] ← S'+D'; M[y] ← M[x]);
```

both righthand sides of the data transmission operator (←) are evaluated in the current memory state in parallel and then transmission occurs. Thus the old value of M[x] would go into M[y]. Serial ordering of processing is indicated by using the term "next". For example,

```
Z := (M[x] ← S'+D'; next M[y] ← M[x]);
```

performs the righthand data transmission after the lefthand one. Thus, the new value of M[x] would be used for M[y] in this latter case.

Memory Declarations

Memory is defined by giving a memory declaration as shown in Table 1. For example,

```
Mp[0:2^k - 1]<15:0>
```

declares a memory named, Mp, of 2^k words (where k has been given a value). The addresses of the words in memory are $0,1,\ldots,2^k-1$. Each word has 16 bits and the bits are labeled 15,14,...,0. Some other examples of memory declarations are:

```
Boundary-error2 }
Boundary-error  }      boolean memories; scalar bit alternatives
Activity3              ternary digit, holding value 0,1, or 2
N/Negative             alias, N and Negative are synonomous
CC<3>                  bit 3 of a register
M[0:2^18-1]<7:0>       vector of 2^18 8-bit words
M[0:15][0:4095]<7:0>   array of 16 x 4096 8-bit words
brop<1:0>16 }          alternative ways of defining a register
brop<7:0>2  }           using base 16 and base 2
```

Renaming and Restructuring of Previously Defined Registers

Registers can be defined in terms of existing registers. In effect, each time the name to the left of the := symbol is encountered, the value is computed according to the expression to the right of :=. A process can be evoked to form the value and side-effects are possible when the value is computed.

Examples of simple renaming in part or whole of existing memory

```
N/Negative := CC<3>          N is name of bit 3 of register CC
SP<15:0> := R[6]<15:0>       SP is the same as register R[6]
```

Examples of register formed by concatenation

```
LAC<L,0:11> := LC□AC<0:11>
AB<0:47> := A<0:23>□B<0:23>
Mword[0]<15:0> := Mbyte[0]<7:0>□Mbyte[1]<7:0>
```

Examples of values and registers formed by evaluation of a process

```
ai/address-increment<1:0> := (   value of ai is 2 if ¬ byte op,
    ¬ byte-op ⇒ 2;                else value is 1
      byte-op ⇒ 1)
Run := (Activity = 0)            Run=1 or 0 depending on value of Activity
                                  being 0 or not 0
```

Instruction Format

Instruction formats are declared in the same fashion as memory and are not distinguishable as special non-memory entities. The instructions are carried in a register; thus it is natural to declare them by giving names to the various parts of the instruction register. Usually only a single declaration is made, the instruction/i, followed by the declarations of the parts of the instruction; the operation code, the address fields, indirect bit, etc.

Example

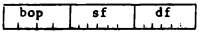
This declaration would correspond to the usual box diagram:

Table 1. ISP Character-Set and Expression Forms

A,...,Z,a,...,z,.,-,␣, ',",0,...,9	name alphabet. This character set is used for names.
	comments. Italics are used for comments.
$M\underbrace{[a:b]\ \ldots\ [v:w]}_{n}\langle x:y\rangle_z$	memory declaration. An n-dimensional memory array of words where a:b ... v:w are the range of values for the first and last dimensions. The values of the first dimension are, for example, a, a+1, ..., b for $a \leq b$ (or a,a-1,...,b for $a \geq b$). The word length base, z, is normally 2 if not specified. The digits of the word are x,x+1,...y.
a := f(expression)	definition. The operator, :=, defines memory, names, process, or operations in terms of existing memory and operations. Each occurrence of "a" causes the in place substitution by f(expression).
b(c,...,e) := g(expression)	The definition b, may have dummy parameters, c,...,e, which are used in g(expression).
name' := h(expression)	side effects naming convention. In this description we have used ' to indicate that a reference to this name will cause other registers to change.
a ← f(expression) f(expression) → a	transmission operator. The contents in register a are replaced by the value of the function.
( )	parentheses. Defines precedence and range of various operations and definitions (roughly equivalent to begin, and end).
{data-type}	operator and data-type modifier
boolean ⇒ expression;	conditional expression; equivalent to ALGOL if boolean then expression
boolean ⇒ (expression-1 else expression-2);	equivalent to Algol if boolean then expression-1 else expression-2
; next	sequential delimiter interpretation is to occur
□	concatenation. Consider the registers to the left and right of □ to be one.
;	statement delimiter. Separates statements.
,	item delimiter. Separates lists of variables.
a/b	division and synonym. Used in two contexts: for division and for defining the name, a, to be an alias (synonym) of the name, b.
?	unknown or unspecified value
ϕ	set value. Takes on all values for a digit of the given base, e.g., $1\phi_2$ specifies either 10_2 or 11_2
X(:= boolean) ⇒ expression;	instruction value definition. The name X is defined to have the value of the boolean. When the boolean is true, the expression will be evaluated.

Common Arithmetic, Logical and Relational Operators

Arithmetic	Logical	Relational
+ add	¬ not	≡ identical
- subtract, also negative	∧ and	≢ not identical
x multiply	∨ or	= equal
/ divide	⊖ exclusive-or	≠ not equal
mod modulo (remainder)	≡ equivalence	> greater than
$( )^2$ squared		≥ greater than or equal
$( )^a$ exponentiation		< less than
( )↑a exponentiation		≤ less than or equal
$( )_b$ base		
( )↓b base		
sqrt( ) square root		
abs( ) absolute value		
sign-extend( )		



i/instruction<15:0>	*the instruction*
bop<3:0> := i<15:12>	*specifies binary (dyadic) operations*
sf<5:0> := i<11:6>	*specifies source (first) operand*
df<5:0> := i<5:0>	*specifies second operand and destination*

Operand Address Calculation Process

In all processors, instructions make use of operands. In most conventional processors, the operand is usually in memory or in the processor, defined as M[z], where z is the effective address. In PDP-11, a destination address, Daddress, is used in this fashion for only two instructions. It is defined in ISP by giving the process that calculates it. This process may involve only accesses to primary memory (possibly indexed), but it may also involve side effects, i.e., the modification of either of primary memory or processor memory (e.g., by incrementing a register). Note that the effective address is calculated whenever its name is encountered in evaluating an ISP expression (either in an instruction or in the interpretation expression). That is, it is evaluated on demand. Consequently, any side effects may be executed more than once.

Operation Determination Processes

Instead of effective-address, the operands are usually determined directly. For example, the 16-bit destination register is just the register selected by the dr field of an instruction, i.e.,

```
Rd := R[dr]                    the destination register
```

In one other case, the operand is just the next word following an instruction. This next word can be defined,

```
nw'<15:0>/next-word := (Mw[PC]; PC ← PC + 2)   the next word is selected and PC is moved
```

Here, the ' shows that a reference to nw will cause side effects, in this case, PC ← PC + 2. For calculating the source operand, S, the process is:

```
S'<15:0> := (                  value for source operand
  (sm=0) ⇒ R[sr];              if mode=0 then S' is the Register addressed
                                by instruction field sr
  (sm=1) ⇒ Mw[R[sr]]           if mode=1 the S' is indirect via R sr
  (sm=2) ∧ (sr=7) ⇒ nw;        if mode=2 and source register=PC then the
                                next word is the operand; this can be
                                seen by substituting the expression for nw'
```

An expression is also needed for the operand, S, which does not cause the side effects, and assuming the effects have taken place, counteracts them. Thus, S would be:

```
S<15:0> := (
   (sm=0) ⇒ R[sr];                    no side effects
   (sm=1) ⇒ Mw[R[sr]];                no side effects
   (sm=2) ∧ (sr=7) ⇒ Mw[PC-2]         counteract previous side effects
      :
      :
```

In the ISP description a general process is given which determines operands for Source-Destination, word-byte, and with-without side-effects. In order to clarify what really happens, the source operand calculation, for words, with side effects, is given below.

```
Sf<5:0> := i<11:6>                    source field (6-bits) of instruction
   sm8 := sf<5:3>                     source mode control field
   sd  := sf<3>                       deferred address control
   sr8 := sf<2:0>                     register specification for source

nw'<15:0> := (Mw[PC]; PC ← PC+2)      next word; used as operand
Rs<15:0>  := R[sr]                    source register specification

S'<15:0>/Source := ((                 value for the source--direct addressing
   (sm=0) ⇒ Rs,                       use the register Rs as operand
   (sm=2) ∧ (sr≠7) ⇒ (Mw[Rs]          direct auto-increment (increment
      Rs ← Rs + 2);                    Rs); usually used as POP
   (sm=2) ∧ (sr=7) ⇒ nw;              direct; actually immediate operand
   (sm=4) ⇒ (Rs ← Rs - 2; next        direct; auto-decrement (decrement
      Mw[Rs]);                         Rs); usually used as PUSH
   (sm=6) ∧ (sr≠7) ⇒ Mw[nw' + Rs];    direct; indexed via Rs--uses next-word
   (sm=6) ∧ (sr=7) ⇒ Mw[nw' + PC];    direct; relative to PC; uses next-word
                                       value for the source-defined addressing
   (sm=1) ⇒ Mw[Rs];                   defer through Rs
   (sm=3) ∧ (sr≠7) ⇒ (Mw[Mw[Rs]];     defer through stack; auto
      Rs ← Rs + 2);                    increment
   (sm=3) ∧ (sr=7) ⇒ M[nw'];          defer via next word; absolute addressing
   (sm=5) ⇒ (Rs ← Rs - 2; next        defer through stack after auto
      Mw[Mw[Rs]]);                     decrement
   (sm=7) ∧ (sr≠7) ⇒ Mw[Mw[nw' + Rs]];  defer, indexed via Rs
   (sm=7) ∧ (sr=7) ⇒ Mw[Mw[nw' + PC]]   defer relative to PC
                  );                   end calculation process;
   (sr=6) ∧ ((sm=4) ∨ (sm=5)) ∧        checks if stack overflowed for several
      (SP<400₈)  ⇒ (Stack overflow ← 1)  modes
                  )                    end source calculation
```

Data-Types

A data-type specifies the encoding of a meaning into an information medium. The meaning of the data-type (what it designates or refers to) is called its referent (or value). The referent may be anything ranging from highly abstract (the uninterpreted bit) to highly concrete (the payroll account for a specific type of employee).

Every data-type has a carrier, into which all its component data-types can be mapped. The carrier is used in storing the data-type in memories and is usually a word or multiple thereof. It must be extensive enough to hold all the component data-types, but may be a larger (having error checking and correcting bits, or

even unused bits). The mapping of the component data-types into the carrier is called the format. It is given as a list which associates to each component an expression involving the carrier (e.g., as in the instruction format).

ISP provides a way of naming data-types, which also serves as a basis for abbreviations. Some data-types simply have conventional names (e.g., character/ch, floating point numbers/f); others are named by their value (e.g., integer/i). Data-types which are iterates of a basic component can be named by the component suffixed by a length-type. The length-type can be array/a, implying a multi-dimensional array of fixed, but unspecified dimensions; a string/st, implying a single sequence, of variable length (on each occurrence); or a vector/v, implying a one dimensional array of a fixed but unspecified number of components. The length-type need not exist, and then this form of the name is not applicable. Thus, iv is the abbreviation for an integer vector. It is also possible to name a data-type by simply listing its components.

Data-types are often of a given precision and it has become customary to measure this in terms of the number of components that are used, e.g., triple precision integers. In ISP this is indicated by prefixing the precision symbol to the basic data-type name, e.g., di for double precision integer. Note that a double precision integer, while taking two words, is not the same thing as a two integer vector, so that the precision and the length-type, though both implying something about the size of the carrier, do not express the same thing.

A list of common data-types and their abbreviations is given in Table 2.

Operations on Data-types

Operations produce results of specific data-types from operands of specific data-types. The data-types themselves determine by and large the possible operations that apply to them. No attempt will be made to define the various operations here, as they are all familiar. A reasonably comprehensive list is given in Table 1. An operation-modifier, enclosed in braces, { }, can be used to distinguish variant operations. The operation-modifier is usually the name of a data-type, e.g., A+B{f} is a floating point addition. Modifiers can also be a description name applying to the operation, e.g., a x 2 {rotate}.

New operations can be defined by means of forms. For example, the various add operations on differing data-types are specified by writing {data-type} after the operation.

Instruction Interpretation Process

The instruction interpretation expression and the instruction set constitute a single ISP expression that defines the processor's action. In effect, this single expression is evaluated and all the other parts of the ISP description of a processor are evoked as indirect consequences of this evaluation. Simple interpreter without interrupt facilities show the familiar cycle of fetch-the-instruction and execute-the instruction.

Example:

```
Run ⇒ (instruction ← M[PC]; PC ← PC + 1; next      This is a simple
        Instruction-execution; next)                interpreter, not the
                                                    one for the PDP-11
```

In more complex processors the conditions for trapping and interrupting must also be dexcribed. The effective address calculation may also be carried out in the interpreter, prior to executing the instruction, especially if it is to be calculated only once and will have a fixed value independent of anything that happens while executing instructions. Console activity can also be described in the interpreter, e.g., the effect of a switch that permits stepping through the program under manual control, or interrogating and changing memory.

The normal statement for PDP-11 interpretation is just:

```
¬ Interrupt-rq ∧ Run ⇒ (instruction ← Mw[PC]; PC ← PC + 2; next   fetch
     Instruction-execution; next                                  execute
     T-flag ⇒ (State-change(14₈); T-flag ← 0))                    trace mode
```

Table 2. Common Data-Types Abbreviations

Primitive		String and Vector	
b	bit or boolean	bv	bit.vector
by	byte	by.st	byte.string
ch	character	ch.st	character.string
cx	complex		
df	double precision floating		
dw	double word		
d	digit	jd	j-digit number
f	floating		
fr	fraction		
hw	half word		
i	integer		
mx	mixed number		
qw	quadruple length word		
tw	triple length word		
w	word		

Instruction-Set and Instruction Execution Process

The instruction set and the process by which each instruction is executed are usually given together in a single definition; this process is called Instruction-execution in most ISP descriptions. This usually includes the definition of the conditions for execution, i.e., the operation code, value, the name of the instruction, a mnemonic alias, and the process for its execution. Thus, an individual instruction typically has the form:

```
MOV (:= ↳op = 0001₂) ⇒ (                    move word
   r ← S'; next                              move source to intermediate register
     N ← r<15>;                              negative?
    (r<15:0> = 0) ⇒ (Z ← 1 else Z ← 0);      zero?
     V ← 0;                                  overflow cleared
     D ← r);                                 transmit result to destination
```

With this format for the instruction, the entire instruction set is simply a list of all the instructions. On any particular execution, as evoked by the interpretation expression, typically one and only one operation code correlation will be satisfied, hence one and only one instruction will be executed.

In the case of PDP-11, the text carries the definition of the individual instructions, hence they are not redefined in the appendix. Instead, the appendix defines the condition for executing the instructions. For example,

```
MOV := (bop = 0001₂)
```

is given in the appendix, and the action of MOV is defined (in ISP) in the text.

PDP-11's Primary (Program) Memory and Processor State

The declaration of this memory includes all the state (bits, words, etc.) that a program (programmer) has access to in this part of the computer. The console is not included. The various secondary memories (e.g., disks, tapes) and input-output device state declarations are included in a following section.

Primary (program) Memory

```
Mp[0:2^k-1]<15:0>                          ! actual physical, 16-bit memory of a particular system; k = 12, ..., 17

  Mw/Mword[x<15:0>]<15:0> := (              ! word-accessed memory
    ¬ x<0> ⇒ Mp[x<15:1>];                  ! word on even byte boundary, all right
      x<0> ⇒ (?value ; Boundary-error ← 1)) ! word on odd byte boundary, trap

  Mb/Mbyte[x<15:0>]<7:0> := (              ! byte-accessed memory
    ¬ x<0> ⇒ Mp[x<15:1>]<7:0>;             ! take low-order bits if even
      x<0> ⇒ Mp[x<15:1>]<15:8>)            ! take hi-order bits if odd
```

Processor State

```
R[0:7]<15:0>                               ! eight, 16-bit General-Registers, used for accumulators, indexing and stacks

  SP<15:0>/Stack-Pointer := R[6]           ! special stack, controlled by R[6]
  PC<15:0>/Program-Counter := R[7]         ! location next instruction, also R[7]

PS<15:0>/Processor-State-Word              ! 16-bit register giving rest of state
  Unused<7:0>/Undefined := PS<15:8>        ! mapping of bits into PS
  P<2:0>/Priority       := PS<7:5>         ! interrupt level control of processor
  T/Trace               := PS<4>           ! denotes whether trap is to occur after each instruction

  CC<3:0>/Condition-Codes := PS<3:0>       ! set as a function of instruction and results
    N/Negative          := CC<3>           ! if result = -
    Z/Zero              := CC<2>           ! if result = 0
    V/Overflow          := CC<1>           ! if result overflows
    C/Carry             := CC<0>           ! if result carried into/borrowed from most significant bit
```

Processor-Controlled Error Flags (resulting from instruction-execution)

```
Boundary-Error                             ! set if word is accessed on odd byte boundary
Stack-Overflow                             ! set if word accessed, via SP < 400(8)
Time-Out-Error                             ! set if non-existent memory or device is referenced
Illegal-Instruction                        ! set if a particular class of instructions is executed
```

Processor-activity

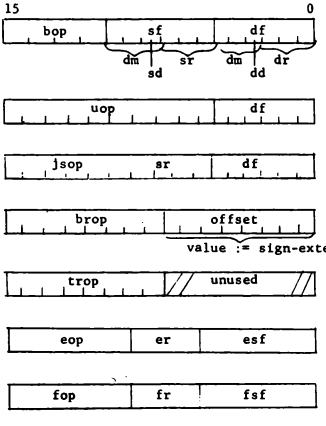
```
  Activity(3)                              ! ternary, specifying state of processor
    Run  := (Activity = 0)                 ! normal instruction interpretation
    Wait := (Activity = 1)                 ! waiting for interrupt
    Off  := (Activity = 2)                 ! off, dormant
```

Error-Flags (resulting from without the processor)

```
Power-Fail-Flag                            ! set if power is low
Power-Up-Flag                              ! set when power comes on
```

Instruction format field declarations

```
i<15:0>/instruction
   bop<3:0>        := i<15:12>          binary opcode format
   sf<5:0>         := i<11:6>           source field
      sm8          := sf<5:3>           source mode - 3 bits
         sd        := sf<3>             source defer bit
      sr8          := sf<2:0>           source register - 3 bits
   df<5:0>         := i<5:0>            destination field
      dm8          := df<5:3>           destination mode - 3 bits
         dd        := df<3>             destination defer bit
      dr8          := df<2:0>           destination register - 3 bits

   uop<3:0>8       := i<15:6>           unary op code (arith., logical, shifts)
   df                                   see binary op format

   jsop<7:0>       := i<15:9>           jsr format
     sr; df                             see binary op format

   brop<1:0>16     := i<15:8>           branch format
   offset<7:0>     := sign-extend(i<7:0>)  offset value

   trop<1:0>16     := i<15:8>           trap format
   unused-trop<1:0>16 := i<7:0>

   eop<6:0>        := i<15:9>           extended opcode format
   er<3:0>         := i<8:6>            extended register
   esf<5:0>        := i<5:0>            extended source field
      esm8         := esf<5:3>          mode
         esd       := esf<3>            defer
      esr8         := esf<2:0>          register

   fop<7:0>        := i<15:8>           floating op format
   fr<7:0>         := i<7:6>            register destination
   fsf<5:0>        := i<5:0>            source
```



binary operand (2 operands) format

unary operand (1 operand), JMP format

JSR format

branch format

trap format

extended operation format

floating op format

```
ai/address-increment<1:0> := (
   ¬ Byte-op ⇒ 2;
     Byte-op ⇒ 1)
Byte-op := (MOVB ∨ BICB ∨ BISB ∨ BITB ∨ CLRB ∨
            COMB ∨ INCB ∨ DECB ∨ NEGB ∨ ADCB ∨
            SBCB ∨ TSTB ∨ RORB ∨ ROLB ∨ ASRR ∨
            ASLB ∨ SWAB)
Reserved-instruction := ((i =  ) ∨ (i =  ) ∨...∨(i =  ))   unused instructions
```

Registers and Data Addressed via Instruction Format Specifications

```
nw/next-word<15:0>  :=  Mw[PC]                          used in operand determination
nw'/next-word'<15:0>:=  (Mw[PC]; PC ← PC + 2)           with side effects
lw/last-word<15:0>  :=  Mw[PC - 2]                      undoes side effects
Rs<15:0>  :=  R[sr]<15:0>                               the source register
Rd<15:0>  :=  R[dr]<15:0>                               the destination register
```

Operand Determination for Source and Destination

Two types of operands are used: S', D', Sb' and Db' - for operands that cause side-effects (i.e., other registers are changed; and S, D, Sb and Db for operands that do not cause side effects. Two general procedures Wo' and Wo are used to determine these operands for side effects and no side effects, respectively

```
S'<15:0>  :=  Oprd'<15:0>(Mw, 2,sm,sr)                  source word operand side-effects
S<15:0>   :=  Oprd<15:0>(Mw, 2,sm,sv)                   source word operands no side-effects
Sb'<7:0>  :=  Oprd'<7:0>(Mb, 2, sm,sr)                  source byte
Sb<7:0>   :=  Oprd<7:0>(Mb, 1,sm,sr)
D'<15:0>  :=  Oprd'<15:0>(Mw, 2,dm,dr)                  Destination operands
D<15:0>   :=  Oprd<15:0>(Mw, 2,dm,dr)
Db'<7:0>  :=  Oprd'<7:0>(Mb, 1, dm,dr)
Db<7:0>   :=  Oprd<7:0>(Mb, 1, dm,dr)
```

General Operand Calculation Process (with Side-Effects)

```
Oprd'<wl:0>(M,ai,m,rg) := ((                            value for word or byte operand; direct
                                                         addressing: wl indicates length; m
                                                         mode, and rg register
   Rr<15:0> := R[rg]                                    secondary definition for register
   (m=0) ⇒ Rr<wl:0>;                                    0, use the register, Rr, as operand
   (m=2) ∧ (rg≠7) ⇒ (M[Rr]; next                        2, direct auto-increment (increment
      Rr ← Rr + ai);                                    Rr); usually used in pop stack
   (m=2) ∧ (rg=7) ⇒ nw'<wl:0>;                          2, direct; next-word is immediate
                                                         operand
   (m=4) ⇒ (Rr ← Rr - ai; next                          4, direct; after auto decrement
      M[Rr]);                                           usually used as PUSH stack
   (m=6) ∧ (rg≠7) ⇒ M[nw' + Rr];                        6, direct; indexed via Rr uses next-
                                                         word
   (m=6) ∧ (rg=7) ⇒ M[nw' + PC];                        6, direct; relative to PC; uses next-
                                                         word value for word operand defer
                                                         addressing
   (m=1) ⇒ M[Rr];                                       1, defer through Rr
   (m=3) ∧ (rg≠7) ⇒ (M[Mw[Rr]]; next                    3, defer through Mw[Rr] (usually stack),
      Rr ← Rr + 2);                                     auto-increment
   (m=3) ∧ (rg=7) ⇒ M[nw'];                             3, defer via next-word; absolute
                                                         addressing
   (m=5) ⇒ (Rr ← Rr - ai; next                          5, defer through stack after auto
      M[Mw[Rr]]);                                       decrement
```

```
   (m=7) ∧ (rg≠7) ⇒ M[Mw[nw' + Rr]];                    ?, defer indexed via Rr
   (m=7) ∧ (rg=7) ⇒ M[Mw[nw' + PC]];                    ?, defer relative to PC
                  );                                    end calculation process
   (rg=6) ∧ ((m=4) ∨ (m=5)) ∧                           check if stack overflows
      (SP < 400₈)) ⇒ (Stack-overflow ← 1)
                  )                                     end operand calculation process
```

General Operand Calculation Process (without Side Effects)

```
Oprd<wl:0>(M,ai,m,rg) := (
   Rr<15:0>  :=  R[rg]
   (m=0) ⇒ Rr.wl:0>;
   (m=2) ∧ (rg≠7) ⇒ Mw[Rr - ai];                        undo previous side-effects
   (m=2) ∧ (rg=7) ⇒ lw.wl:0>;                           undo previous side-effects
   (m=4) ⇒ M[Rr];
   (m=6) ∧ (rg≠7) ⇒ M[lw + Rr];                         undo previous side-effects
   (m=6) ∧ (rg=7) ⇒ M[lw + PC];                         undo previous side-effects

   (m=1) ⇒ M[Rr];
   (m=3) ∧ (rg≠7) ⇒ M[Mw[Rr - 2]];                      undo previous side-effects
   (m=3) ∧ (rg=7) ⇒ M[lw];                              undo previous side-effects
   (m=5) ⇒ M[Mw[Rr]];
   (m=7) ∧ (rg≠7) ⇒ M[Mw[lw + Rr]];                     undo previous side-effects
   (m=7) ∧ (rg≠7) ⇒ M[Mw[lw + PC]])                     undo previous side-effects
```

Destination addresses for JMP and JSR

```
Da<15:0>  := ((                                         directs:
   (dm=0) ⇒ (?; Illegal-instruction ← 1);                  illegal register address
   (dm=2) ∧ (dr≠7) ⇒ (Rd; Rd ← Rd + 2);                    auto-increment
   (dm=2) ∧ (dr=7) ⇒ (PC; PC ← PC + 2);                    null
   (dm=4) ⇒ (Rd ← Rd - 2; next Rd);                        auto-decrement
   (dm=6) ∧ (dr≠7) ⇒ (nw' + Rd);                           indexed
   (dm=6) ∧ (dr=7) ⇒ (nw' + PC);                           relative
                                                        defers:
   (dm=1) ⇒ Mw[Rd];                                        via register
   (dm=3) ∧ (dr≠7) ⇒ (Mw[Rd]; Rd ← Rd + 2);                via auto-increment
   (dm=3) ∧ (dr=7) ⇒ nw';                                  absolute address
   (dm=5) ⇒ (Rd ← Rd - 2; next Mw[Rd]);                    auto-decrement
   (dm=7) ∧ (dr≠7) ⇒ Mw[nw + Rd];                          via index
   (dm=7) ∧ (dr=7) ⇒ Mw[nw' + PC]); next                   relative to PC

   (dr=6) ∧ ¬ ((dm=0) ∨ (dm=3) ∨ (dm=7)) ∧ (SP < 400₈) ⇒ ( check for stack overflow
      stack-overflow ← 1))
```

Data Type Formats

```
by/byte<7:0>
w/word<15:0>
wi/word.integer<15:0>
bybv/byte.boolean-vector<7:0>
wbv/word.boolean-vector<15:0>
d/d.w/double.word<31:0>
```

```
f/d.f/double.word.floating<31:0>
   fs/floating.sign := f<31>
   fe/floating.exponent<7:0> := f<30:23>
   fm/floating.mantissa<22:0> := f<22:0>
t/triple.word<47:0>
q/quadruple.word<63:0>
qf/quadruple.word.floating-point<63:0>
   qfs := qf<63>
   qfe := qf<62:55>
   qfm := qf<54:0>

I/O Devices and Interrupts, State Information
Device[0:N-1]                                          N I/O devices - assume device J
   Device-name[J]<15:0> := J                           number to which device responses and
                                                        is addressed
   Device-interrupt-location[J]<15:0> := K             each device has a value, K, which it
                                                        uses as an address to interrupt processor
   dob/device-output-buffer[J]<15:0>                   program controlled device data
   dib/device-input-buffer[J]<15:0>
   ds/device-status[J]<15:0>                           a register with device control state
      derr/device-error-flags[J]<3:0> := ds[J]<15:12>  common
      dbusy/device-busy[J] := ds[J]<11>                status
      dunit/device-unit-selection[J]<2:0> := ds[J]<10:8> assignments
      ddone[J] := ds[J]<7>
      denb/device-done-interrupt-enable := ds[J]<6>
      derrenb/device-error-interrupt-enable := ds[J]<5>
      dme/device-memory-extension[J]<4:3> := ds[J]<4:3>
      dfnc/device-function[J]<2:0> := ds[J]<2:0>
dintrq/device-interrupt-request[J] := (
   (ddone[J] ∧ denb[J] ∨ ((derr[J] ≠ 0) ∧ derrenb[J]))
   dil/device-interrupt-level[J]<7:4>                  each device is assigned to 1 of 4 levels

Mapping of Devices into M.  Each device's registers are mapped into primary word memory, e.g.,
Teletype
   M'[177560₈] := tks/ds[TTY-keyboard]                 keyboard status
   M'[177562₈] := tkb/dib[TTY-keyboard]                keyboard input data
   M'[177564₈] := tps/ds[TTY-printer]                  teleprinter status
   M'[177566₈] := tpb/dob[TTY-printer]                 teleprinter data to print

Interrupt Requests
br/bus-request-for-interrupt<7:4> := (                 OR of all device requests
   (dintrq[0] ⇒ dil[0]) ∨
   (dintrq[1] ⇒ dil[1]) ∨...
   (dintrq[J] ⇒ dil[J]) ∨...
   (dintrq[N] ⇒ dil[N]))

Interrupt-rq := (intrql ≥ p)                           interrupt if a request is ≥ priority/P

intrql/interrupt-request-level<2:0> := (
   br<7> ⇒ 7;
   ¬ br<7> ∧ br<6> ⇒ 6;
   ¬ br<7> ∧ ¬ br<6> ∧ ¬ br<5> ∧ br<4> ⇒ 4)
```

```
Instruction Interpretation Process
Interrupt-rq ∧ Run ⇒ (Normal-interpretation);
   Normal-interpretation := (I ← Mw[PC]; PC ← PC + 2 next        fetch
      Instruction-execution; next                                execute
      T-flag ⇒ (State-change(14₈); T-flag ← 0))                  trace
Interrupt-rq ∧ ¬ Off ⇒ (
   State-change(Device-interrupt-location[J]);                   assume device J interrupts
   P ← intrq1);
off ⇒ ( );
¬ Interrupt-rq ∧ Wait ⇒ ( );
   State-change(x) := (                                          for stacking state and restore
      SP ← SP - 2; next
      Mw[SP] ← PS;
      SP ← SP - 2; next
      Mw[SP] ← PC;
      PC ← Mw[x];
      PS ← Mw[x+2]
Boundary-Error ⇒ (State-change(4₈); Boundary-error ← 0)

Time-Out-Error → (State-change(4₈); Time-Out-Error ← 0)
Power-Fail-Flag ⇒ (state-change(24₈); Power-Fail-Flag ← 0;) program must turn off computer
Power-Up-Flag ⇒ (PC ← 24₈; Power-Up-Flag ← 0; Activity ← 0) Start Up on power-up

Instruction-Set Definition
     Each instruction is defined in ISP in the text, therefore, it will not be repeated here.
```

```
ISP for Floating Point Processor/FPP
Device-interrupt-location [FPP] := M'[244₈]
FEC<15:0>                                          floating point processor error code
                                                    register
   FOCE := (FEC=2)                                 floating op code error
   FDZE := (FEC=4)                                 floating divide by zero
   FICE := (FEC=6)                                 floating integer
                                                    conversion error
   FVE  := (FEC=8)                                 floating overflow
   FUE  := (FEC=10)                                floating underflow
   FUVE := (FEC=12)                                floating undefined variable

FAC[0:5]<63:0>                                     6 floating point accumulators
Fr<63:0>                                           temporary floating point register
FPC<15:0>                                          floating point PC
FPSR<15:0>                                         floating point processor status register
   FER := FPSR<15>                                 floating error
   FIE := FPSR<14>                                 interrupt enable
   FIUV:= FPSR<11>                                 interrupt on undefined variable
   FIU := FPSR<10>                                 interrupt on underflow
   FIV := FPSR<9>                                  interrupt on overflow
   FIC := FPSR<8>                                  interrupt on integer conversion error
   FD  := FPSR<7>                                  floating double precision mode
   FL  := FPSR<6>                                  floating long integer mode
   FT  := FPSR<5>                                  floating truncate mode
   FMM := FPSR<4>                                  floating maintenance mode
```

```
   FN  := FPSR<3>                          floating negative
   FZ  := FPSR<2>                          floating zero
   FV  := FPSR<1>                          floating overflow
   FC  := FPSR <0>                         floating carry

Instruction format
OC<3:0>    := i<15:12>                     op code
FOC<3:0>   := i<11:8>                      floating op code
AC<1:0>    := i<7:6>                       accumulator


General Definitions
XL  := ((FD=0) ⇒ 1-2^-24;                  largest fraction
       (FD=1) ⇒ 1-2^-56)
XLL := 2^-128                              smallest non-zero number
XUL := 2^127 * XL                          largest number
JL  := ((FL=0) ⇒ 2^15-1;                   largest integer
       (FL=1) ⇒ 2^31-1)

Address Calculation
FPS<63:0> := (                             floating point processor source
   (dm=0) ⇒ FAC(dr);
   (dm≠0) ⇒ (
      (FD=0) ⇒ D<15:0>□Mw[PC+2];
      (FD=1) ⇒ D<15:0>□Mw[PC+2]□
         Mw[PC+4]□Mw[PC+6]))
FPS'<63:0> := (                            floating point processor source with
   (dm=0) ⇒ FAC(dr);                        side effects
   (dm≠0) ⇒ (
      (FD=0) ⇒ D'<15:0>□nw'
      (FD=1) ⇒ D'<15:0>□nw'□nw'□nw'))
FPD<63:0> := FPS<63:0>                     floating point processor destination
FPD'<63:0>:= FPS'<63:0>                    floating point processor destination with
                                            side effects
FS<15:0>  := D<15:0>                       floating source, CPU mode
FS'<15:0> := D'<15:0>                      floating source with side effects,
                                            CPU mode
FD<15:0>  := D<15:0>                       floating destination, CPU mode
FD'<15:0> := D'<15:0>                      floating destination with side effects,
                                            CPU mode
Fac := FAC(AC)                             destination floating register
```

[1] a 17 bit result, r, used only for descriptive purposes

[2] A prime is used in S (e.g., S') and D (e.g., D') to indicate that when a word is accessed in this fashion, side effects may occur. That is, registers of R may be changed.

[3] If all 16 bits of result, r = 0, then Z is set to 1 else Z is set to 0.

[4] The 8 least significant bits are used to form a 16-bit positive or negative number by extending bit 7 into 15:8.

[5] a ⇒ b means: if boolean a is true then b is executed.

[6] Mw means the memory taken as a work-organized memory.

APPENDIX B

INSTRUCTION TIMING

B.1 EXECUTION TIME

In the PDP-11/45, the execution time for an instruction depends on the instruction itself, the modes of addressing used, and the types of memory referenced. The time is actually the sum of these three independent times: source time, destination time, and the time to execute this instruction and fetch the next. Instructions do not always require all three times to be added and thus are divided into four classes depending on the number of items to be summed.

NOTES

Memory Segmentation

When memory segmentation (KT11) is in use, 90 ns must be added to every memory access.

Interleaved Core:

When the 850 ns memory (MM11S) is interleaved it has an effective cycle time of 650 ns for an external device; however, due to overlap of central processor operation with memory cycles, there is virtually no advantage as far as processor instruction timing is concerned. The tables therefore present the instruction times for Core (850 ns), MOS (450 ns), and Bipolar (300 ns).

B.1.1 Execution and Fetch of Next Instruction (times in ns)
Class I

Instruction Execution = SRC time + DST time + Execution and Fetch of Next Instruction

DST mode = 0

	Memory Cycles	850 ns Core (MM11-S)	450 ns MOS (MS11-B)	300 ns Bipolar (MS11-C)
SWAB [1]	1	850	450	300
CLR(B) [1]	1	850	450	300
COM(B) [1]	1	850	450	300
INC(B) [1]	1	850	450	300
DEC(B) [1]	1	850	450	300
ADC(B) [1]	1	850	450	300
SBC(B) [1]	1	850	450	300
ASR(B) [1]	1	850	450	300
ASL(B) [1]	1	850	450	300
ROR(B) [1]	1	850	450	300
ROL(B) [1]	1	850	450	300
BIC(B) [1]	1	850	450	300
BIS(B) [1]	1	850	450	300
ADD [1]	1	850	450	300
SUB [1]	1	850	450	300
XOR [1]	1	850	450	300
SXT [1]	1	850	450	300
MOVB [1]	1	850	450	300
MOV [1]	1	850	450	300
MFPx	2	2290	1500	1200
MTPx	2	1990	1160	900
NEG(B)	1	1290	900	750
TST(B) [1]	1	850	450	300
CMPB(B) [1]	1	850	450	300
BIT(B) [1]	1	850	450	300
JMP	1		Illegal, Trap will occur	
JSR	2		Illegal, Trap will occur	

[1] Add 290 ns core, 450 ns for MOS, 300 ns for Bipolar if DST = R7

Class I (continued)

DST mode ≠ 0

	Memory Cycles	850 ns Core (MM11-S)	450 ns MOS (MS11-B)	300 ns Bipolar (MS11-C)
SWAB	1	1620	940	750
CLR(B)	1	1620	940	750
COM(B)	1	1620	940	750
INC(B)	1	1620	940	750
DEC(B)	1	1620	940	750
ADC(B)	1	1620	940	750
SBC(B)	1	1620	940	750
ASR(B) [2]	1	1620	940	750
ASL(B)	1	1620	940	750
ROR(B) [2]	1	1620	940	750
ROL(B)	1	1620	940	750
BIC(B)	1	1620	940	750
BIS(B)	1	1620	940	750
ADD	1	1620	940	750
SUB	1	1620	940	750
XOR	1	1620	940	750
SXT	1	1620	940	750
MOVB	1	1620	940	750
MOV	1	850	450	300
MFPx	2	2140	1350	1050
MTPx	2	1840	1010	750
NEG(B)	1	1830	1350	1050
TST(B)	0	1140	640	450
CMP(B)	0	1140	640	450
BIT(B)	0	1140	640	450
JMP	1	1140	750	600
JSR	2	2290	1500	1200

[2] Add 150 ns if DST address is an odd byte

Class II:

Instruction Execution = SRC time + Execution and Fetch of next instruction but SRC time is obtained from DST table

	Memory Cycles[2]	850 ns Core (MM11-S)	450 ns MOS (MS11-B)	300 ns Bipolar (MS11-C)
ASH[3]	1	1440 + 150*shift	1050 + 150*shift	900 + 150*shift
ASHC[3]	1	1440 + 150*shift	1050 + 150*shift	900 + 150*shift
MUL	1	3840	3450	3300
DIV:				
by 0	1	1440	1050	900
min[4]	1	7440	7050	6900
max[4]	1	8040	7650	7500

[3] Less 150 ns if SRC = 0

[4] Plus 900 ns if dividend is negative

[5] 0 if SRC mode not 0.

Class III

Instruction Execution = Execution and Fetch of next instruction

	Memory Cycles	450 ns MOS (MS11-B)	850 ns Core (MM11-S)	300 ns Bipolar (MS11-C)
BR yes	1	1140	900	600
no	1	850	450	300
SOB yes	1	1140	900	600
no	1	1290	900	750
RTI	3	3040	2060	1500
RTT	3	3040	2060	1500
RTS	2	2040	1460	1050
MARK	2	1990	1350	900
Condition Code Ops	1	1140	900	600
SPL	1	1140	900	600
Interrupt	5	5190	3260	2550
Trap	5	4890	3110	2250

Class IV Indeterminate or Long Instructions

Reset	10 ms
Halt	∞
WAIT	Variable

B.1.2 SOURCE TIME

Mode	Memory Cycles		850ns Core (MM11-S)	450ns MOS (MS11-R)	300ns Bipolar (MS11-C)
0	0	0	0	0	0
1	1	840	370	300	
2	1	840	370	300	
3	2	1830	930	750	
4	1	990	560	450	
5	2	1980	1120	900	
6	2	1600	820	600	
7	3	2590	1380	1050	

B.1.3 DESTINATION TIME:

Add 150 ns to times if destination is an odd byte except where SRC mode (where applicable) and DST mode both equal 0, or where DST = R7

Class I JMP, JSR

Mode	Memory Cycles	850 ns core (MM11-S)	450 ns MOS (MS11-B)	300 ns Bipolar (MS11-C)
0	0	Illegal, Trap will occur		
1	1	300	300	300
2	1	300	300	300
3	2	1050	710	600
4	1	300	300	300
5	2	1200	860	750
6	2	1000	560	450
7	3	1990	1120	900

Class II: MOV, MTPx NOT MOVB

Mode	Memory Cycles	850 ns core (MM11-S)	450 ns MOS (MS11-B)	300 ns Bipolar (MS11-C)
0	0	150[1]	190[1]	150[1]
1	1	850[3]	640[2]	450
2	1	850[3]	640[2]	450
3	2	1840[2]	1200[2]	900
4	1	990	750	600
5	2	2080	1310	1050
6	2	1900[1]	1200[1]	900[1]
7	3	2890[1]	1760[1]	1350[1]

Class III: All others

Mode	Memory Cycles	850 ns core (MM11-S)	450 ns MOS (MS11-B)	300 ns Bipolar (MS11-C)
0	0	150[1]	190[1]	150
1	2	760[4]	450[2]	300
2	2	760[4]	450[2]	300
3	3	1660[2]	1010[2]	750
4	2	900[4]	560	450
5	3	1800	1120	900
6	3	1660[1]	1010[1]	750[1]
7	4	2560[1]	1570[1]	1200[1]

[1] Subtract 150 ns for core and bipolar, 190 ns for MOS if SRC mode = 0, or is not applicable.

[2] Subtract 10 ns core, 40 ns MOS if SRC mode = 0, or is not applicable

[4] Add 80 ns if SRC mode = 0, or is not applicable.

Example:

ADD @R1,A(R2) :Add mode 1(R1), mode 6(R2)

	850 ns Core (MM11-S)	450ns MOS (MS11-B)	300 ns Bipolar (MS11-C)
SRC	840	370	300
DST	1660	1010	750
Execution + Fetch	1620	940	750
	4.120 μs	2.320 μs	1.800 μs

B.2 FLOATING POINT INSTRUCTION TIMING

Floating point times are calculated in a similar manner to the CPU Instruction times. The times involved are preexecution Interaction time, source or destination time, execution time, CPU displacement time, and the time taken to fetch the next Instruction.

With the floating point Instructions the CPU and the FPP operate in parallel and hence, the Instruction time includes a CPU time and a parallel FPP time. These times do not coincide, each unit is free to continue at a different time with its next operation.

Instruction Time (CPU) = pre-interaction + source + disengage + fetch of the next instruction.

Instruction Time (FPP) = pre-interaction + source + execution

Pre-execution Interaction time: This involves the passing of information between the CPU and the FPP. The total time is 600 ns. The floating point unit Interacts only during the last 150 ns.

Therefore, the CPU becomes active at the time 0, and remains so until time 600 ns, while the FPP becomes active at time 450 and remains active until the time 600 ns. The FPP could have been active from a previous task during the initial 450 ns.

B.2.1 Source or Destination Times

These times are the same whether the calculation is for source or destination. The times given are for the address calculation. To this must be added memory access time to actually fetch the operands:
Integer - 1 word; Long integer and Floating - 2 words;
Double Precision - 4 words

Therefore: SOURCE/DST = Calculation Time + MEMORY ACCESS TIME.

Calculation Time

	850 ns Core (MM11-S)	450 ns MOS (MS11-B)	300 ns Bipolar (MS11-C)
Reg mode 0	1200	1200	1200
Floating mode 0	600	600	600
1	750	750	750
2	750	750	750
3	1500	1160	1050
4	750	750	750
5	1650	1310	1200
6	1500	1160	1050
7	2400	1720	1500

MEMORY ACCESS TIME: to be added to all modes except 0.

ADD: 1230 ns for core OR
ADD: 710 ns for MOS OR
ADD: 600 ns for BIPOLAR
for every memory Access required to fetch the data.

B.2.2 Floating Point Execution

Floating Point Instructions

Mnemonic	Instruction	Time (μs) MIN	MAX
Floating AC—Floating Source Group: OPR FSRC, AC			
LDF	Load floating	0	0
LDD	Load floating double	0	0
ADDF	Add floating	2.4	3.8
ADDD	Add floating double	3.0	5.4
SUBF	Subtract floating	2.4	3.8
SUBD	Subtract floating double	3.0	5.4
MULF	Multiply floating	3.8	5.8
MULD	Multiply floating double	5.8	11.2
DIVF	Divide floating	3.8	9.2
DIVD	Divide Floating double	5.8	17.6
MODF	Multiply and Integerize floating	5.4	11.6
MODD	Multiply and Integerize floating double	7.6	21.8
LDCDF	Load and convert from double to floating	2.0	2.4
LDCFD	Load and convert from floating to double	2.0	2.0
STF	Store floating	0	0
STD	Store floating double	0	0
Floating AC—Floating Destination Group: OPR AC, FDST			
CMPF	Compare floating	1.2	1.8
CMPD	Compare double	1.6	2.2
STCFD	Store and convert from floating to double	.6	
STCDF	Store and convert from double to floating	1.8	
Floating AC—Source Group: OPR SRC, AC			
LDCIF	Load and convert from integer to floating	2.8	3.8
LDCID	Load and convert from integer to double	3.0	4.0
LDCLF	Load and convert from long integer to floating	2.8	4.6
LDCLD	Load and convert from long integer to double	3.0	4.8
LDEXP	Load Exponent	.8	

Mnemonic	Instruction	Time (μs)	
Floating AC—Destination Group: OPR AC, DST			
		MIN	MAX
STCFI	Store and convert from floating to integer	3.4	4.4
STCFL	Store and convert from floating to long integer	4.2	5.2
STCDI	Store and convert from double to integer	4.2	5.2
STCDL	Store and convert from double to long integer	4.2	5.2
STEXP	Store exponent	1.4	
Floating Destination Group: OPR FDST			
CLRF	Clear floating	.6	
CLRD	Clear double	1.0	
NEGF	Negate floating	1.0	
NEGD	Negate double	1.0	
ABSF	Make absolute floating	.6	
ABSD	Make absolute double	.6	
TSTF	Test floating	.4	
TSTD	Test double	.4	
Operate Group: OPR SRC			
LDFS	Load floating program status OPR DST		0
STFPS	Store floating program status		0
STST	Store floating status (exception & code and program counter		0
CFCC	Copy Condition codes		0
SETF	Set floating mode		0
SETI	Set integer mode		0
SETD	Set double mode		0
SETL	Set long integer mode		0

B.2.3 Disengage and Fetch Next Instruction

This is the time required by the CPU to disengage from the FPP and fetch the next instruction. If the instruction is a Floating Point instruction then the CPU restarts the cycle described in this section (B.2), otherwise, it returns to the CPU cycle described in section B.1.

CPU time only:

990 ns Core

600 ns MOS

450 ns Bipolar

Example

ADDF A(R), AC0 :a floating point add instruction in indexed source mode (6), and in single (32 bit precision).

Pre-Interaction Time:
CPU 600 ns FPP 150 ns

Source:	850 ns Core (MM11-S)	450 ns MOS (MS11-B)	300 ns Bipolar (MS11-C)
Calculation Time:	1500	1160	1050
Memory Access (2 words)	2460	1420	1200
	3960	2580	2250
Execution Average	3100	3100	3100
Disengage Fetch Next	990	600	450

CPU Instruction Time: Starting at T = 0

600 + 3960 + 990	600 + 2580 + 600	600 + 2250 + 450
5.550	3.780	3.300

FPP Instruction Time: Starting at T = 450

150 + 3960 + 3100	150 + 2580 + 3100	150 + 2250 + 3100
4.210	5.830	5.500

B.3 Latency

UNIBUS NPR WORST CASE: 3.5 μs

Interrupt:
(If highest priority)

Worst Case: Negative Divide with SRC mode 7

850 ns Core (MM11-S)	450 ns MOS (MS11-B)	300 ns Bipolar (MS11-C)
11.350 μs	9.930 μs	9.450 μs

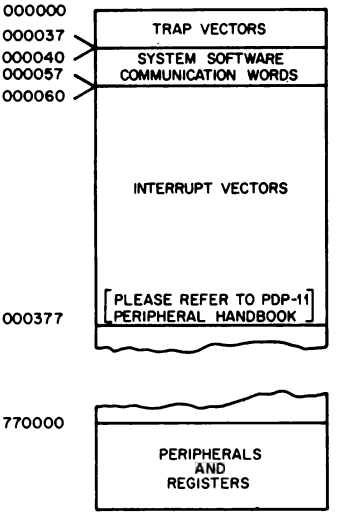
Add 360 ns if segmentation is in use.

Average Case:

3.5 μs	2.0 μs	1.7 μs

APPENDIX C

MEMORY MAP AND RESERVED LOCATIONS



The location of trap vectors for processor conditions is as follows:

Note: all locations are located in Kernal Virtual Address Space.

Vector Condition

4 Odd address error - an attempt has been made to reference a word at an odd address. The instruction causing the error has been aborted (at the micro-instruction causing the "bus pause" associated with the bad bus cycle).

4 Fatal stack violation (red), warning stack violation (yellow) - an attempt has been made to modify (DATIP, DATO, OR DATOB) a word whose address is below the final stack boundary (red), or 16 words before the boundary (yellow) with an R6-related operation.* The instruction causing the error is aborted (as above), R6 is set equal to the quantity 4 and a trap is taken. Odd address errors, segment violations, and timeout associated with R6-related operations will also lead to fatal stack violations (instead of their normal sequence).

* The final stack boundary is 16 words beyond the stack limit.

4 Timeout - no SSYN was seen within 5 μsec of processor transmission of MSYN. The instruction causing the error is aborted.

4 Parity error - incorrect data parity has been detected by a slave device during a processor initiated DATA or DATIP. Note that this will be acted upon only at the next bus cycle initiated by the processor.

10 Illegal and reserved instructions - JMP R;JSR M,R; "HALT"in user mode; FPP instructions when FPP not available; and the reserved instructions are:

000007-000077
000210-000227
007000-007777
075000-076777
106400-107777

14 Opcode 000003 and the T bit trap through this vector. The T bit causes this trap whenever it is set and there is no RTT instruction in the instruction register.

20 IOT Trap

24 Power Fail

30 Emulator Trap (EMT)

34 Trap Instruction (TRAP)

Interrupt Vectors
240 Programming Interrupt Request

244 Floating Point Exception
250 Segmentation violations and segmentation memory management traps

Register Locations
Memory Segmentation:(All Memory Segmentation Registers use 2 Word Locations)

777572 through 777577 Memory Segmentation Status Registers 0-2
772516 through 772517 Memory Segmentation Status Register 3
777600 through 777617 User Instruction Descriptor Registers 0-7
777620 through 777637 User Data Descriptor Registers 0-7
777640 through 777657 User Instruction Address Registers 0-7
777660 through 777677 User Data Address Registers 0-7
772200 through 772217 Supervisor Instruction Descriptor Registers 0-7
772220 through 772237 Supervisor Data Descriptor Registers 0-7
772240 through 772257 Supervisor Instruction Active Registers 0-7
772260 through 772277 Supervisor Data Address Registers 0-7
772300 through 772317 Kernel Instruction Descriptor Registers 0-7
772320 through 772337 Kernel Data Descriptor Registers 0-7
772340 through 772357 Kernel Instruction Address Registers 0-7
772360 through 772377 Kernel Data Address Registers 0-7

Memory Parity Status Registers:

772110 Memory Parity Status Register 0-8K
772112 Memory Parity Status Register 8K-16K

772114 Memory Parity Status Register 16K-24K
772116 Memory Parity Status Register 24K-32K
772120 Memory Parity Status Register 32K-40K
772122 Memory Parity Status Register 40K-48K
772124 Memory Parity Status Register 48K-56K
772126 Memory Parity Status Register 56K-64K
772130 Memory Parity Status Register 64K-72K
772132 Memory Parity Status Register 72K-80K
772134 Memory Parity Status Register 80K-88K
772136 Memory Parity Status Register 88K-96K
772140 Memory Parity Status Register 96K-104K
772142 Memory Parity Status Register 104K-112K
772144 Memory Parity Status Register 112K-120K
772146 Memory Parity Status Register 120K-124K
Processor:
777570 Console Switch and Display Register
777772 Program Interrupt Register
777774 Stack Limit Register
777776 Processor Status Word

ORDER OF SERVICE

In the case of concurrent trap/interrupt conditions. The PDP-11/45 service requests in the following order:

Order	Condition	Action
0	Odd Address	Trap (4)
0	Fatal Stack Violation (Red)	SP4; Trap (4)
0	Segment Violation	Trap (10)
0	Timeout (NXM)	Trap (4)
0	Parity Error	Trap (4)
1	FPP Data XFER Request	DO Bus Cycle
2	Console Flag	Console Control
3	Segment Management Trap	Trap (250)
4	Warning Stack Violation (Yel)	Trap (4)
5	Power/Fail	Trap (24)
	Processor Priority Level 7	
6	FPP Exception Trap	Trap (224)
7	PIRQ 7	Trap (240)
8	BR7,INTR	Interrupt
	Processor Priority Level 6	

9	PIRQ 6	Trap (240)
10	BR6,INTR	Interrupt
	Processor Priority Level 5	
	Processor Priority Level 1	
17	PIRQ 1	Trap (240)
	Processor Priority Level 0	
18	T-Bit Set and not RTT	Trap (14)

NOTES

1. Order 0 conditions are mutually exclusive and take immmediate precedence over all other conditions. They force immediate interruption (abort) of instruction execution when they occur.
2. Other order conditions permit the instruction in progress (including traps) to proceed to completion before being serviced.
3. Lower order conditions depend on the Processor Status setup by previously serviced conditions. Thus power/fail service should (and generally would) raise the processor priority to level 7 to lock out interrupts.
4. The T-bit condition is not locked out by any processor priority level.
5. Fatal Stack Violations superceed warning stack violations. In case both occur during an instruction only the fatal violation is acknowledged.
6. If a Segment Management Trap and a Fatal Stack Violation occur together, the PDP-11/45 will loop:

   A. The Fatal Stack Violation sequence is exceeded.

   B. The first instruction of the "Trap4"Service Routine is executed setting the SP to 0.

   C. The Segment Management Trap begins but causes a Fatal Stack Violation.

   D. Step A, Step B, Step C,...

7. If power fails before execution of a Fatal Stack Violation Trap Sequence:

   A. The PS and PC are taken from the power fail sequence

   B. The SP is set to 4.

C. The previous PC and PS (from the routine causing the stack violation) are saved and (pushed into) locations 0 and 2.

D. Execution of the Power/Fail Service Routine begins - a suggested beginning is:

```
TST SP
BNE SPOK
MOV #N,SP      ;set up emergency stack
```

8. If power failed during (or after) execution of a Fatal Stack Violation Trap Sequence:

   A. The Stack Violation Trap Sequence completes - saving the PC and PS associated with the violation in locations 0 and 2 and establishing a new PS and PC as determined by the vector at 4.

   B. The first instruction of the "Trap 4" service routine is executed.

   C. The Power/Fail Vector is loaded into the PS and PC.

   D. An attempt is made to push the old PC and PS into the stack. The stack pointer however is probably at 0 so a Fatal Stack Violation occurs.

   E. Operation proceeds as in note #7. The PC and PS of the routine causing the Fatal Stack Violation are overwritten. (they were saved in locations 0 and 2). The PC and PS for the "Trap 4" Service Routine are the same as the contents of the Power/Fail Vector - also the same as the contents of the current PS and PC.

The vectors for Power/Fail and Stack Violation Traps (at 24 and 4) must raise the processor Priority to level 7 to lock out interrupts. In addition they must not set the T-bit. Since there is no stack left, any interrupts or T-Bit Traps will cause Fatal Stack Violation and either:

   A. Loop if the vector at 4 is mis-set

   B. Loss of the Power/Fail Trap if the vector at 24 is mis-set.

10. If an interrupt or trap causes a Fatal Stack Violation, the PS and PC loaded from the Interrupt Vector are saved in locations 0 and 2. The PC and PS of the Interrupted (Main Line) Program are lost.

11. If any order 0 conditions occur in the Fatal Stack Violation Trap Sequence before a new stack is established. The processor will enter the loop described in Note 6 above.

12. If a particular error condition is endangered by the routine to service that condition the stack will grow to its limit and then cause a Fatal Stack Violation

13. Reserved and illegal instructions (including the Floating Point instructions when Floating Point Hardware is not implemented as well as attempts to halt in User Mode) are treated just as instruction traps (e.g., trap, EMT). Note however that in the case of Floating Point Instructions the PC pushed on the stack points at the instruction word.

PROGRAM INTERRUPT REQUESTS

Program Interrupt Requests
A request is booked by setting one of the bits 15 through 9 (for PIR 7 - PIR 1) in the Program Interrupt Register at location 777772. The hardware sets bits 7 - 5 and 3 - 1 to the encoded value of the highest PIR bit set. This Program Interrupt Active (PIA) should be used to set the Processor Level and also index through a table of interrupt vectors for the seven software priority levels. Figure D-1 shows the layout of the PIR Register.

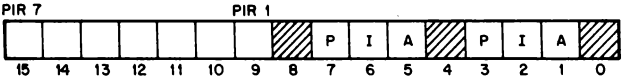


Figure D-1 Program Interrupt Request Register

When the PIR is granted, the Processor will Trap to location 240 and pick up PC in 240 and the PSW in 242. It is the interrupt service routine's responsibility to queue requests within a priority level and to clear the PIR bit before the interrupt is dismissed.

The actual interrupt dispatch program should look like:

```
MOVB   PIR,PS          ; places Bits 5 - 7 in PSW Priority Level
                         Bits
MOV    R5,-(SP)        ; save R5 on the stack
MOV    PIR,R5
BIC    #177761,R5      ; Get Bits 1 - 3
JMP    @ DISPAT(R5)    ; use to index through table
```

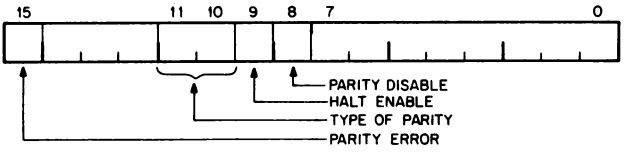
which requires 15 core locations.

APPENDIX E

Memory Parity

When memory parity is installed on a system, the user can keep track of the status of his memory through memory status registers.

There are 16 memory status registers on the PDP 11/45 each one associated with an 8K section of memory. (See Appendix C) The status register uses the format:



Bit 15	Parity Error	
Bits 11,10	Type of Parity	Bits 11 and 10 are associated with the high order 4K and low order 4K of this memory address bank. When set to a 1 they specify odd parity for their respective half banks; when clear, even parity.
Bit 9	Halt Enable	When bit 9 is set the machine will execute a halt if a parity error occurs; when clear, the machine will perform an effective timeout and 'interrupt' through location 4.
Bit 8	Parity Disable	When this bit is clear, a parity error will cause an interrupt (or halt as specified in bit 9); if it is set, no action will be taken on a parity error.

When the machine is powered-up, the status registers have bit 15 set to 0, and the remaining bits set to 1: halt,odd parity enable; parity disable.

INDEX

Absolute Addressing....37
Addressing 3,25
Architecture....9
Autodecrement Mode 31
Autoincrement Mode 29
Branch Instructions....81
Bus Options....5
Byte Instructions 46
Central Processor....2, 11
Co-Routines....146
Communications 6
Core Memory 16
Condition Codes 14
Condition Code Instructions...130
Console 197
CPU 2, 11
Data Acquisition and Control 7
Deferred Addressing....34
Direct Addressing 28
Direct Memory Access 4
DMA 4
Double Operand Instructions....68
Floating Point Instructions....173
Floating Point Processor4,14,167
FORTRAN 6
FPP 4
General Registers 12
Immediate Mode....37
Index Mode....32
Indirect Addressing 34
Instructions....2,43
Instruction Timing....219
Instruction Set Processor ..44,205
Interleaving 17
ISP....44,205
Logical Instructions....77
Memories....4,14
Memory Management....147
Memory Parity....17
Memory Retention 16
Memory Segmentation 4,147
Modes, Addressing 25
Nesting....140
Operator's Console....147
Options....5
PC 26
PDP-11 Family....1
Peripherals 5
PIC 145
Position Independent Code 145
Power Fail And Restart 4
Power Failure....22
Priority Interrupts....3,19
Processor Status Word 13,24
Processor Traps 22
Processor Priority....13
Program Counter....26,36
Program Control Instructions...81
Programming Techniques 131
PS....13,24
Recursion 146
Reentrancy 143
Reentrant Code 3
Register Mode 28
Registers 12
Relative Addressing....38
Rotate Instructions....64
RSTS-11 6
Segmentation....4,147
Shift Instructions....55
Single Operand Instructions....47
Software....6
Solid State Memory....15
SP....25
Stack Limit Register 14
Stack Pointer....25
Stacks 4,14,131
Storage Devices....5
Subroutine Instructions....102
Subroutines....135
System Interaction 18
T Bit 14
Timing....219
Traps....14,22
Trap Instructions....112
UNIBUS....1,10

INSTRUCTION INDEX

ADC(B) 61
ADD 71
ASL(B) 57
ASH 58
ASHC 59
ASR(B) 56

BCC 89
BCS 88
BEQ 84
BGE 94
BGT 96
BHI 98
BHIS 101
BIC(B) 80
BIS(B) 78
BIT(B) 79
BLT 93
BLE 95
BLO 100
BLOS 99
BMI 86
BNE 85
BPL 87
BPT 115
BR 82
BVC 91
BVS 90

CLR(B) 48
CMP(B) 73
COM(B) 53

COND. CODES 130
DEC(B) 49
DIV 75

EMT 113

HALT 123

INC(B) 50
IOT 116

JMP 110
JSR 103

MARK 105
MFPD 129
MFPI 128
MOV(B) 69
MTPD 127
MTPI 126
MUL 74

NEG(B) 51

RESET 125
ROL(B) 65
ROR(B) 66
RTI 117
RTS 107
RTT 118

SBC(B) 62
SOB 111
SPL 109
SUB 72
SWAB 67
SXT 63

TRAP 114
TST(B) 52

WAIT 124

XOR 76

ABSD....190
ABSF....190
ADDD....176
ADDF....176

CFCC....195
CLRD....189
CLRF....189
CMPD....181
CMPF....181

DIVD....180
DIVF....180

LDCDF....183
LDCFD....183
LDCID....185
LDCIF....185
LDCLD....185
LDCLF....185
LDD....174
LDEXP....187
LDF....174
LDFPS....194

MODD....182
MODF....182
MULD....179
MULF....179

NEGD....178
NEGF....178

SETD....192
SETF....192
SETI....193
SETL....193
STCDF....184
STCDI....186
STCDL....186
STCFD....184
STCFI....186
STCFL....186
STD....175
STEXP....188
STF....175
STFPS....194
STST....195
SUBD....177
SUBF....177

TSTD....191
TSTF....191

NOTES

NOTES

NOTES

DIGITAL EQUIPMENT CORPORATION digital WORLD-WIDE SALES AND SERVICE

MAIN OFFICE AND PLANT

146 Main Street, Maynard, Massachusetts, U.S.A. 01754 • Telephone From Metropolitan Boston 646-8600 • Elsewhere: (617)-897-5111
TWX 710-347-0212 Cable: DIGITAL MAYN Telex 94-8457

UNITED STATES

NORTHEAST

REGIONAL OFFICE
15 Lunda Street, Waltham, Massachusetts 02154
Telephone (617)-891-1030/1033 TWX 710-324-6919

WALTHAM
15 Lunda Street, Waltham, Massachusetts 02154
Telephone: (617)-891-8310/6315 TWX 710-324-6919

CAMBRIDGE/BOSTON
899 Main Street, Cambridge, Massachusetts 02139
Telephone (617) 491-6130 TWX 710-320-1167

ROCHESTER
130 Allens Creek Road, Rochester, New York 14618
Telephone (716) 461-1700 TWX 710-253-3078

CONNECTICUT
240 Pomeroy Ave., Meriden, Conn. 06450
Telephone (203) 237-8441 7466 TWX 710 461-0054

MID-ATLANTIC — SOUTHEAST

REGIONAL OFFICE
U.S. Route 1, Princeton, New Jersey 08540
Telephone (609)-452-2940 TWX 510-685-2338

NEW YORK
95 Cedar Lane, Englewood, New Jersey 07631
Telephone (201) 871-4984, (212) 594-6955, (212)-736-0447
TWX 710-991-9721

NEW JERSEY
1259 Route 46, Parsippany, New Jersey 07054
Telephone (201) 335-3300 TWX 710-987-8319

PRINCETON
U.S. Route 1
Princeton, New Jersey 08540
Telephone (609) 452-2940 TWX 510-685-2338

LONG ISLAND
1919 Middle Country Road
Centereach, L.I., New York 11720
Telephone (516)-585-5410/5413 TWX: 510-228-6505

PHILADELPHIA
Station Square Three, Paoli, Pennsylvania 19301
Telephone: (215)-647-4800/4410 Telex 510-668-8395

WASHINGTON
Executive Building
6811 Kenilworth Ave., Riverdale, Maryland 20840
Telephone: (301)-779-1800/752-8797 TWX 710-826-9682

DURHAM/CHAPEL HILL
2704 Chapel Hill Boulevard
Durham, North Carolina 27707
Telephone (919)-489-3347 TWX 510-927-0912

ORLANDO
Suite 232, 6990 Lake Ellenor Drive, Orlando, Fla. 32809
Telephone (305)-851-4450 TWX 810-850-0180

ATLANTA
2815 Clearview Place, Suite 100,
Atlanta, Georgia 30340
Telephone: (404)-458-3133/3134/3135 TWX 810-757-4223

MID-ATLANTIC — SOUTHEAST (cont.)

KNOXVILLE
5731 Lyons View Pike, S.W., Knoxville, Tenn. 37919
Telephone (615)-588-6571 TWX 810-583-0123

CENTRAL

REGIONAL OFFICE:
1850 Frontage Road, Northbrook, Illinois 60062
Telephone (312)-498-2500 TWX 910-686-0655

PITTSBURGH
400 Penn Center Boulevard
Pittsburgh, Pennsylvania 15235
Telephone (412)-243-9404 TWX 710-797-3657

CHICAGO
1850 Frontage Road, Northbrook, Illinois 60062
Telephone (312)-498-2500 TWX 910-686-0655

ANN ARBOR
230 Huron View Boulevard, Ann Arbor, Michigan 48103
Telephone (313) 761-1150 TWX 810-223-6053

INDIANAPOLIS
21 Beachway Drive — Suite G
Indianapolis, Indiana 46224
Telephone (317)-243-8341 TWX 810-341-3436

MINNEAPOLIS
Suite 111, 8030 Cedar Avenue South,
Minneapolis, Minnesota 55420
Telephone (612)-854-8562-3-4-5 TWX 910-576-2818

CLEVELAND
Park Hill Bldg., 35104 Euclid Ave
Willoughby, Ohio 44094
Telephone (216)-946-8484 TWX 810-427-2808

ST. LOUIS
Suite 110, 115 Progress Pky., Maryland Heights,
Missouri 63043
Telephone (314)-872-7520 TWX 910-764-0831

DAYTON
3101 Kettering Blvd., Dayton, Ohio 45439
Telephone (513) 299-7377 TWX 810-459-1676

MILWAUKEE
Suite 107, 2625 N. Mayfair Rd., Milwaukee, Wis. 53222
Telephone (414)-453-3400 TWX 910-262-1199

DALLAS
8855 North Stemmons Freeway
Dallas, Texas 75247
Telephone (214)-638-4880 TWX: 910-861-4000

HOUSTON
3417 Milam Street, Suite A, Houston, Texas 77002
Telephone (713)-524-2961 TWX 910-881-1651

CENTRAL (cont.)

NEW ORLEANS
3100 Ridgelake Drive, Suite
Metairie, Louisiana 70002
Telephone 504-837-0257

WEST

REGIONAL OFFICE
1400 Terra Bella, Mountain View, California 94040
Telephone (415)-964-6200 TWX 910-373-1266

ANAHEIM
801 E. Ball Road, Anaheim, California 92805
Telephone (714)-776-6932/8730 TWX 910-591-1169

WEST LOS ANGELES
1510 Cotner Avenue, Los Angeles, California 90025
Telephone (213)-479-3791/4318 TWX 910-342-6999

SAN DIEGO
3444 Hancock Street
San Diego, California
TWX 910-335-1230

SAN FRANCISCO
1400 Terra Bella
Mountain View, California 94040
Telephone (415)-964-6200 TWX: 910-373-1266

PALO ALTO
760 San Antonio Rd., Palo Alto, California 94306
Telephone (415)-969-6200 TWX 910-373-1266

OAKLAND
7810 Edgewater Drive
Oakland, California 94621
Telephone (415)-635-5453/7830 TWX 910-366-7238

ALBUQUERQUE
6303 Indian School Road, N.E.
Albuquerque, N.M. 87110
Telephone (505)-298-5411/5428 TWX: 910-989-0614

DENVER
2305 South Colorado Blvd., Suite #5
Denver, Colorado 80222
Telephone (303)-757-3332/758-1656/758-1659
TWX 910-931-2650

SEATTLE
1521 130th N.E., Bellevue, Wash. 98005
Telephone (206)-454-4058/455-5490 TWX: 910-443-2306

SALT LAKE CITY
431 South 3rd East, Salt Lake City, Utah 84111
Telephone: (801)-328-9838 TWX: 910-925-5834

PHOENIX
307 E. Southern Ave., Tempe, Arizona 85281
Telephone: (602)-967-1618 TWX: 910-950-4981

PORTLAND
5319 S.W. Westgate Dr., Portland, Ore. 97221
Telephone (503) 297-3761/3765

INTERNATIONAL

EUROPEAN HEADQUARTERS

Digital Equipment Corporation International Europe
81 Route de l'Aire
1211 Geneva 26, Switzerland
Telephone 42 79 50 Telex 22 683

FRANCE

Equipment Digital S.A.R.L.
PARIS
327 Rue de Charenton, 75 Paris 12 EME, France
Telephone 344-78-07 Telex: 21339

SPAIN

SR Ruiz Escudero (Sales Only)
Ataio Ingenieros S.A.
Enrique Larreta, 12
Madrid 16, Spain

GERMANY

Digital Equipment GmbH
MUNICH
8 Muenchen 13, Wallensteinplatz 2
Telephone 0811-35031 Telex: 524-226
COLOGNE
5 Koeln, Bismarckstrasse 7,
Telephone 0221-522181 Telex 888-2269
Telegram: Flip Chip Koeln
FRANKFURT
6078 Neu-Isenburg 2
Am Forsthaus Gravenbruch 5-7
Telephone 06102-5526 Telex 41-76-82
HANNOVER
3 Hannover, Podbielskistrasse 102
Telephone 0511-69-70-95 Telex: 922-952

UNITED KINGDOM

Digital Equipment Co., Ltd.
U.K. HEADQUARTERS
Arkwright Road, Reading, Berks
Telephone 0734-583555 Telex 84327
READING
The Evening Post Building, Tessa Road
Reading, Berks
BIRMINGHAM
29-31, Birmingham Road, Sutton Coldfield, Warwicks
Telephone (0044) 21-355 5501 Telex 337 060
MANCHESTER
13 Upper Precinct, Walkden, Manchester
Telephone 061-790-8411 Telex 668666
LONDON
Bilton House, Uxbridge Road, Ealing, London W.5
Telephone 01-579-2334 Telex: 22371

NETHERLANDS

THE HAGUE
Digital Equipment N.V.
Sir Winston Churchillaan 370
Rijswijk The Hague, Netherlands
Telephone 070-995-180 Telex: 32533

BELGIUM

BRUSSELS
Digital Equipment N.V./S.A.
108 Rue D'Arlon
1040 Brussels, Belgium
Telephone: 02-139256 Telex: 25297

SWEDEN

Digital Equipment Aktiebolag
STOCKHOLM
Vretenvagen 2, S-171 54 Solna, Sweden
Telephone: 98 13 90 Telex: 170 50
Cable: Digital Stockholm

NORWAY

Digital Equipment
OSLO
c/o Firma Service
Waldenmarthranesgate 84 B-86
Oslo 1, Norway
Telephone: 37 19 85, 37 02 30 Telex: 166 43

SWITZERLAND

Digital Equipment Corporation S.A.
GENEVA
81 Route de l'Aire
1211 Geneva 26, Switzerland
Telephone 42 79 50 Telex 22 683
ZURICH
Scheuchzerstrasse 21
CH-8006 Zurich, Switzerland
Telephone 01 60 35 66 Telex 56059

ITALY

Digital Equipment S.p.A.
MILAN
Corso Garibaldi 49, 20121 Milano, Italy
Telephone: 872 748 694 384 Telex: 33815

ISRAEL

Digital Equipment Corp.
TEL AVIV
America House, 33 Shaoel Hamelecha
Tel Aviv, Israel
Telephone (03) 25 23 41 Telex 033 139

AUSTRALIA

Digital Equipment Australia Pty. Ltd.
SYDNEY
P.O. Box 491, Crows Neet
N.S.W. Australia 2065
Telephone: 439-2566 Telex: AA20740
Cable: Digital, Sydney
MELBOURNE
60 Park Street, South Melbourne, Victoria, 3205
Telephone: 69-8142 Telex: AA30700
WESTERN AUSTRALIA
643 Murray Street
West Perth, Western Australia 6005
Telephone: 21-4993 Telex: AA92140
BRISBANE
139 Merivale Street, South Brisbane
Queensland, Australia 4101
Telephone 44404? Telex: AA40616
NORWOOD
6 Montrose Avenue
Norwood, S.A., Australia 5067
Telephone: 63-1399

CANADA

Digital Equipment of Canada, Ltd.
CANADIAN HEADQUARTERS
150 Rosamond Street, Carleton Place, Ontario
Telephone: (613)-257-2815 TWX: 610-561-1651
OTTAWA
120 Holland Street, Ottawa 3, Ontario
Telephone (613)-725-2193 TWX: 610-562-8907
TORONTO
230 Lakeshore Road East, Port Credit, Ontario
Telephone (416)-278-6111 TWX: 610-492-4306
MONTREAL
9675 Cote de Liesse Road
Dorval, Quebec, Canada 760
Telephone: 514-636-9393 TWX: 610-422-4124
EDMONTON
5531-103 Street
Edmonton, Alberta, Canada
Telephone (403)-434-9333 TWX: 610-831-2248
VANCOUVER
Digital Equipment of Canada, Ltd.
2210 West 12th Avenue
Vancouver 9, British Columbia, Canada
Telephone: (604)-736-5616 TWX: 610-929-2006

ARGENTINA

BUENOS AIRES
Coasin S.A.
Virrey del Pino 4071, Buenos Aires
Telephone: 52-3185 Telex: 012-2284

VENEZUELA

CARACAS
Coasin S.A. (Sales only)
Apartado 50939
Salana Grande No. 1, Caracas
Telephone 72-9637 Cable: INSTRAUVEN

CHILE

SANTIAGO
Coasin Chile Ltda. (sales only)
Casilla 14588, Correo 15, Santiago
Telephone 396713 Cable COACHIL

JAPAN

TOKYO
Rikei Trading Co., Ltd. (sales only)
Kozato-Keikan Bldg.
No. 18-14, Nishishimbashi 1-chome
Minato-Ku, Tokyo, Japan
Telephone: 5915246 Telex: 781-4206
Digital Equipment Corporation International
Kowa Building No. 17, Second Floor
2-7 Nishi-Azabu 1-Chome
Minato-Ku, Tokyo, Japan
Telephone: 404-5894 6 Telex: TK-8428

PHILIPPINES

Stanford Computer Corporation
P.O. Box 1608
416 Dasmarinas St., Manila
Telephone 49-88-96 Telex: 742-0352

digital

digital equipment corporation

112 • 01071 • 1876
D-09-15

Printed in U.S.A.

pdp11/45